N-63-4-3

ASD-TDR-63-280

MODULAR ARITHMETIC COMPUTING TECHNIQUES

Technical Documentary Report No. ASD-TDR-63-280

MAY 1963

Electronics Technology Division
Aeronautical Systems Division
United States Air Force
Wright-Patterson Air Force Base, Ohio

Project No. 7062, Task No. 706205

(Prepared under Contract No. AF33(657)-7899 by the
Westinghouse Electric Corporation, Air Arm Division,
Baltimore, Maryland.)

36-888N, 200, 7-22-63

FOREWORD

This report was prepared by Westinghouse Electric Corporation, Baltimore, Maryland, on Air Force Contract AF33(657)7899, Project 7062, Task 706205. The work was administered under the direction of the Electronic Technology Laboratory, Aeronautical Systems Division, Wright-Patterson Air Force Base, Ohio, Mr. D. J. Boaz, Project Engineer.

The contractor's report number is 1274A.

This is the final report on the contract.

ABSTRACT

Modular arithmetic concepts and associated computation techniques and organization are outlined. Fundamental operations of modular arithmetic discussed include sign or relative magnitude determination and division, mathematical solution using modular arithmetic, techniques for efficient mechanization of modular arithmetic adders and multipliers, and organization and control of a modular arithmetic computer.

Numerical analysis studies yielded novel results including the introduction of signed residues, overflow detection techniques, and a division algorithm 3 times as fast as any previously disclosed. A square-root algorithm which is considerably faster than the Newton-Raphson algorithm is discussed.

Implementation techniques are described for trading speed for complexity, programming a computer to extend its range, and design techniques for reducing component counts in adders and multipliers.

A functional simulation of modular arithmetic computation in a conventional computer is described, and statistical data on the operation of the various algorithms is included.

REVIEW AND APPROVAL STATEMENT

This report represents the specific findings of an Air Force sponsored program. It does not direct any specific application thereof. The report is approved for publication to achieve an exchange and stimulation of ideas.

FOR THE COMMANDER

Leonard M. Butsch, Jr.

LEONARD M. BUTSCH, JR.
Lt Colonel, USAF
Chief, Bionics Branch
Electronic Technology Division

TABLE OF CONTENTS

5. SIMULATION OF MODULAR ARITHMETIC COMPUTERS ON THE IBM 7090

8. REFERENCES

LIST OF ILLUSTRATIONS

LIST OF TABLES

GLOSSARY

ASM	Add, subtract, or multiply control signal
FAP	Fortran assembly program
GF	Galois field
LAVR	Least absolute value residue
LNVR	Least nonnegative value residue
MA	Modular arithmetic
TSMR	"Two-sided" mixed radix

1. INTRODUCTION AND SUMMARY

This report contains the results of an approximately 1-year study of modular arithmetic concepts and associated computation techniques and organization. Areas of investigation included the "difficult" fundamental operations of modular arithmetic; namely, sign or relative magnitude determination and division, the solution of various mathematical problems using modular arithmetic, techniques for efficient mechanization of modular arithmetic adders and multipliers, and the organization and control of a modular arithmetic computer.

Novel results of the numerical analysis studies include the introduction of signed residues and the collateral "two-sided" mixed radix representation, overflow detection techniques, and a division algorithm more flexible than, and almost 3 times as fast as, any previously disclosed. Studies of mathematical problems yielded a square-root algorithm which is considerably faster (for modular arithmetic computation) than adapting the conventional Newton-Raphson algorithm and an efficient technique for overcoming a problem peculiar to modular arithmetic in using the powerful technique of Gauss elimination.

Implementation studies, frequently capitalizing on the results of analytic studies of modular relations, have yielded techniques for time sharing of logic elements to permit a designer to trade speed for complexity or for programming a computer to extend its range, as well as several design techniques which lead to a significantly reduced component count in adders and multipliers. Several specific sample implementations were derived to obtain accurate gate counts.

A functional simulation of modular arithmetic computation on a conventional computer was developed and used as an aid to analysis and to provide statistical data on the operation of various algorithms.

The next section summarizes significant conclusions which may be drawn from the results of this study. Section 3 collects recommendations for further study which are described in greater detail in several referenced paragraphs of this report.

Manuscript released by author for publication February 1963 as an ASD Technical Documentary Report.

Section 4 presents the results of the numerical analysis studies, both of fundamental operations and of applications to mathematical problems. Section 5 describes the simulator and its application.

Section 6 describes a possible modular arithmetic computer organization and control specification. It also compares modular arithmetic and conventional computers. Finally, Section 7 presents techniques and results of implementation studies.

2. CONCLUSIONS

The modular arithmetic technique should be considered as a serious competitor to conventional arithmetic techniques in any fixed-point computer. This general conclusion is derived from the specific conclusions which follow and which, in turn, are drawn from the study results presented in Sections 4, 6, and 7:

a. A broad, almost continuous spectrum of tradeoff between complexity of mechanization and speed of operation exists for modular arithmetic computers.

b. A typical modular arithmetic computer using presently available numerical algorithms and techniques for minimizing the implementation complexity of modular arithmetic would have the following speeds contrasted to a comparable conventional fixed-point computer of reasonable range:

Addition and subtraction - twice as fast

Multiplication - ten times as fast

Division - one-third as fast

Sign determination - five to ten times slower

c. A modular arithmetic computer can efficiently use fixed radix subroutines; however, special subroutines which minimize the use of division and/or sign determination can increase the efficiency of the computer. A specific example is the square-root algorithm of paragraph 4.1.6.

d. The most economical mechanization in terms of computations/unit cost is by means of solid state logic elements.

e. The mechanization producing greatest speed is with semiconductor logic elements.

f. The use of signed residues increases speed of certain operations and reduces the logic element count ot the arithmetic units.

3. RECOMMENDATIONS

On the basis of the studies and results described in detail later in this report, several study areas are recommended. The present state of the art in modular arithmetic is developed enough that further studies of a general nature on fundamental algorithms are not recommended. However, the problem of sign or magnitude determination is so important in itself and in several key algorithms that specific promising ideas should be investigated.

On the other hand, studies of implementation and of techniques for minimizing the cost of implementing modular adders and multipliers are highly recommended. Such studies should be oriented to capitalize on properties peculiar to modular arithmetic. On the basis of studies of this type reported herein, considerable progress can be made in this most important area.

Specific study recommendations follow. In each case, details of the items listed can be found in the referenced paragraphs of this report.

a. The use of continued fraction approximations to find 2 small integers whose ratio is a good approximation to the ratio of 2 large integers. Paragraph 4.3.1.

b. The use of a cyclic orientation algorithm in ordering lists of numbers and in magnitude comparison. Paragraph 4.3.2.

c. The generation of efficient table look-up procedures for modular arithmetic computers. Paragraph 4.3.3.

d. Optimum memory addressing techniques for modular arithmetic computers. Paragraph 6.5.2.

e. The study of an optimum instruction repertoire and of hypothetical computer configurations in solving specific problems through the use of a classical simulator. Paragraph 6.5.2.

f. The use of mod m arithmetic units to perform mod k arithmetic to increase system reliability or range for special problems. Paragraph 6.5.2.

g. The special properties of the Boolean functions of modular arithmetic as a subclass of the class of 2^{2^n} Boolean functions of n variables. Paragraph 7.3.2.

h. The further use of multiphase techniques to simplify mechanization at only an insignificant decrease in speed. Paragraph 7.3.2.

i. The sharing of logic between adders and multipliers to reduce total arithmetic unit complexity. Paragraph 7.3.2.

j. The use of isomorphisms between various residue groups as a means of determining low cost implementations for large moduli as well as for sharing of hardware as in i. Paragraph 7.3.2.

k. The use of pseudo-single moduli to decrease mixed radix conversion time both separately and in conjunction with wire-twisting techniques. Paragraph 7.3.2.

4. NUMERICAL ANALYSIS

4.1 FUNDAMENTAL OPERATIONS

4.1.1 Sign and Magnitude Determination (Two-Sided Mixed Radix Notation)

It is well known that any integer, x, may be written in mixed radix notation,

$$x = a_1 + a_2 m_1 + a_3 m_1 m_2 + \dots + a_n m_1 m_2 \dots m_{n-1}$$

where $0 \leq a_i < m_i$, $i = 1, 2, \dots, n$, and $0 \leq x < m_1 m_2 \dots m_n$. The radices, m_i, need not be distinct. If the a_i are permitted to take on negative, as well as positive, values, the notation is called "two-sided" mixed radix notation (TSMR). In this case it is required that the a_i be of least absolute value; that is, $|a_i| \leq (m_i - 1)/2$ for odd m_i and $|a_i| \leq m_i/2$ for even m_i. If m_i is even and $|a_i| = m_i/2$, a_i may be either positive or negative, but not both. If only m_1 is even, it can be shown that the above TSMR notation determines a unique x, such that $|x| \leq m_1 m_2 \dots m_n/2 = M/2$. However, if some other m_i $(i > 1)$ is even, $|x|$ may not be less than $M/2$. When $|x| = M/2$ (and only m_1 is even), $x = M/2$ or $-M/2$ according as a_1 is interpreted as positive or negative when $|a_1| = m_1/2$.

A principal merit of TSMR is that it provides a simple means of determining the sign of a number, x, given in residue form and of comparing the magnitude of x with that of another number. For, it can be shown readily that the magnitude of any nonzero term in the TSMR expression above is greater than the sum of all terms to its left in the expression. If follows from this that, if a_k is the rightmost nonzero coefficient in the above expression, then

$$(a_k - \tfrac{1}{2}) K \leq x \leq (a_k + \tfrac{1}{2}) K,$$

where

$$K = m_1 m_2 \dots m_{k-1}.$$

Obviously, then, the sign of x is the same as that of a_k. Furthermore, if

$$y = b_1 + b_2 m_1 + b_3 m_1 m_2 + \dots + b_n m_1 m_2 \dots m_{n-1}, \tag{1}$$

is the TSMR expression for y, $x > y$ if $a_n > b_n$ and $x < y$ if $a_n < b_n$. If $a_n = b_n$, $x > y$ or $x < y$ according as a_{n-1} is greater or less than b_{n-1}, etc. Proceeding in this manner gives the expected result that $x = y$ if and only if $a_i = b_i$, $i = 1, 2, \ldots n$.

In order for TSMR to be of any practical value, a means must be provided to convert a number, x, in residue form to TSMR form. If the moduli, m_i, are relatively prime and the residues are given in least absolute value form, it follows from equation 1 above that $y = y_1 \equiv b_1 \pmod{m_1}$ and $(y_{i-1} - b_{i-1})/m_{i-1} \equiv b_i \pmod{m_i}$, $1 < i \leq n$. Since the m_i are relatively prime, we can eliminate them in succession from these n equations and obtain the coefficients, $b_1, b_2, \ldots, b_n$.

For example, if we permit only the residues 0 and 1 (mod 2), the moduli, $m_1 = 2$, $m_2 = 3$, $m_3 = 5$, $m_4 = 7$, will allow unique representation of all integers x such that $-104 \leq x \leq 105$. Then, $103 \rightarrow (1, 1, -2, -2)$ produces the following array as m_1, m_2, and m_3 are eliminated:

	mod 2	3	5	7
y_1	$b_1 = \underline{1}$	1	-2	-2
y_2		$b_2 = \underline{0}$	1	2
y_3			$b_3 = \underline{2}$	3
y_4				$b_4 = \underline{3}$.

Hence, $103 = 3(2 \cdot 3 \cdot 5) + 2(2 \cdot 3) + 0(2) + 1$.

If the "inverses" of the m_i (that is, s_{ij} such that $s_{ij} m_i \equiv 1 \pmod{m_j}$, $i < j$) are stored in a table, any division by an m_i in the above process can be performed as a multiplication by the corresponding set of s_{ij}. Thus, the entire elimination process can be completed using only (n - 1) subtractions and (n - 1) multiplications for the n moduli, $m_1, m_2, \ldots, m_n$. Since $|b_i| \leq m_i/2$, the coefficients, $b_1, b_2, \ldots, b_n$, can be stored as a set of residues.

Conversion from TSMR back to residue form may be effected by evaluating the expression, $a_1 + a_2 m_1 + a_3 m_1 m_2 + \ldots + a_n m_1 m_2 \ldots m_{n-1}$. If the n moduli are available from an n-word permanently stored table (word 1 = m_1, word 2 = m_2, etc), the (purely modular) conversion again requires (n - 1) multiplications and (n - 1) additions. However, unless the moduli are arranged in ascending order ($m_1 < m_2 < \ldots < m_n$), it is also necessary to reconvert the a_i to residue form by routing them through the input decimal-to-modular converters.

4.1.2 Overflow Detection

In the following paragraphs the problem of overflow detection is discussed. In paragraph 4.1.2.1, a relatively simple method of detecting additive overflow is presented. In paragraph 4.1.2.2, a somewhat tedious method of detecting multiplicative overflow is described, while in paragraph 4.1.2.3 a more rapid solution to this problem is given. This solution, however, requires a small stored table which is also used in the processes described in paragraphs 4.1.5, 4.1.6, and 4.1.7. Finally, in paragraph 4.1.2.4, overflow detection of the result of perhaps a complicated sequence of calculations is discussed. Here, no attempt is made to detect overflow of a single calculation; indeed, if such overflow occurs but is later cancelled by a further calculation of the sequence, it will, as is desired, not be detected.

4.1.2.1 Additive Overflow

The problem to be considered here is how to detect, by means of modular arithmetic operations on the addends and sum, additive overflow when it occurs. Two properties of such overflow are immediately obvious: First, at most one overflow can occur, since $0 \leq |a| < M$ and $0 \leq |b| < M$ imply that $0 \leq |a| + |b| < 2M$, where M is the machine range; second, overflow cannot occur if a and b have opposite signs because in that case, $0 \leq |a + b| = ||a| - |b|| \leq \max(|a|, |b|) < M$.

For the remaining cases, if a and b have the same sign and $c \equiv |a + b| \pmod M$, then overflow occurs (that is, $|a| + |b| \geq M$) if and only if $c < |a|$ (or $c < |b|$); for, if no overflow occurs, $c = |a + b| = |a| + |b| \geq |a|$. Hence, $c < |a|$ implies that overflow has occurred. Conversely, if (one) overflow occurs, then

$$c = |a| + |b| - M < |a|, \text{ since } |b| - M < 0.$$

If "two-sided" residue notation is used, both $|a|$ and $|b| \leq M/2$, so that $|a| + |b| \leq M$ and overflow ($|a| + |b| > M/2$, in this case) can be detected by the presence of the "wrong" sign on $c' \equiv a + b \pmod M$.

4.1.2.2 Multiplicative Overflow - Method I

The problem of overflow detection in multiplication is considerably more difficult than in addition because more than one "overflow" can occur in a modular arithmetic multiplication; that is, $|x \cdot y|$ may be greater than 2M. Consider the following examples in which "two-sided" residues are used with the moduli 2, 3, 5, and 7:

$78 \cdot 103 \equiv (0, 0, -2, 1) \cdot (1, 1, -2, -2)$

$= (0, 0, -1, -2) \equiv 54 \equiv 8034 \pmod{210}$;

$9 \cdot 29 \equiv (1, 0, -1, 2) \cdot (1, -1, -1, 1)$

$= (1, 0, 1, 2) \equiv 51 \equiv 261 \pmod{210}$.

If $z \equiv |x| \cdot |y| \pmod M$ is such that $|z| = |x|$, then either $x = 0$ or $|y| = 1$, and similarly for $|z| = |y|$. Therefore, if $|z| = |x|$ or $|z| = |y|$, overflow cannot occur. And, as in addition, if z has the "wrong" sign, or if $|z| < |x|$ or $|z| < |y|$, then overflow has occurred. But, if $|z| > |x|$ and $|z| > |y|$, overflow may occur, as in the multiplication of 9 by 29 above, or may not occur, since $|z| = |x \cdot y| > |x|$ and $|x \cdot y| > |y|$, if $|x| \cdot |y| < M$ and both $|x|$ and $|y| > 1$.

In the case where $|z| > |x|$ and $|z| > |y|$, we proceed as follows. Let k be a positive integer such that $k^2 \leq [\text{machine range}] < (k + 1)^2$, and assume for convenience that $|x| \geq |y|$. Define a and b by $a = |x| - k$ and $b = |y| - k$. Obviously, if $a \leq 0$ and $b \leq 0$, then $|x| \cdot |y| \leq k^2$, and overflow does not occur. Likewise, if $a > 0$ and $b > 0$, then $|x| \cdot |y| \geq (k + 1)^2$, and overflow does occur. Here, and in the remainder of this section, we say that overflow occurs if and only if $|x| \cdot |y| > [\text{machine range}]$. If "two-sided" residues are used, the machine range is different for positive and negative integers. This complicates matters somewhat, but with a few modifications, the following method is still valid.

In the case where $a > 0$ and $b \leq 0$, let us examine the expression, $|x| \cdot |y| = (k + a) \cdot (k + b) = k^2 + (a + b) k + ab$. If we define $d \geq 0$ such that $d = [\text{machine range}] - k^2$, then $|x| \cdot |y|$ overflows if and only if $(a + b) k + ab > d$. If we rearrange $(a + b) k + ab$ to give $a(b + k) + bk = a|y| + bk$ and note from the definition of b that $0 \leq -b = |b| < k$ and $|y| \leq k$, it follows that $a \leq |b|$ implies that $a|y| \leq |b| \cdot |y| \leq |b| \cdot k$. Therefore, if $a \leq |b|$, then $(a + b) k + ab = a|y| + bk \leq 0 \leq d$, and overflow does not occur.

Let us consider the remaining case - that for which $a > |b|$ and $a > 0 \geq b$. It follows from the definition of b that $(a + b) k + ab + b^2 = (a + b) \cdot (k + b) = (a + b) |y|$. Therefore, if $(a + b) |y| > d + b^2$, overflow occurs. But, $|b| < k$, so that $d + b^2 < d + k^2 = [\text{machine range}]$. Hence, if $(a + b) |y|$ overflows, it is certainly greater than $d + b^2$; otherwise, we can compare $(a + b) |y|$ with $d + b^2$ to determine whether or not $|x| \cdot |y|$ overflows.

Now only one problem remains - that of determining whether or not $(a + b) |y|$ overflows. For this, we set $(a + b) = x_1 > 0$, and repeat the entire procedure for

$x_1 |y|$. It may be necessary to do this several times, thus making this overflow detection method an iterative procedure.

Let us summarize this method by defining

$x_0 = x$, $x_i = |x_{i-1}| + b - k$, $a_i = |x_i| - k$, and $z_i \equiv |x_i| \cdot |y|$ (mod M).

It follows that if any $x_i |y| = (a_{i-1} + b) |y|$ overflows for $i > 0$, $|x| \cdot |y|$ overflows. That is, if $b > 0$ or if $z_i < |x_i|$ for any i, then $|x| \cdot |y|$ overflows; and if $|z_0| = |x_0| \cdot |z_0| = |y_0|$, or $a_0 \leq b$, then $|x| \cdot |y|$ does not overflow. Otherwise, for some $i > 0$, $|z_i| = x_i$ or $z_i = |y|$ or $a_i \leq |b|$ (since $a_i \leq a_{i-1} -1$, this must happen eventually), and overflow occurs if and only if $(a_0 + b) |y| = x_1 |y| = z_1 > d + b^2$.

Thus, we can detect multiplicative overflow with the above method, but several "iterations" may be required in certain cases. Such instances probably occur quite seldom, since the test, $z_i < |x_i|$ or $z_i < |y|$, is usually sufficient to detect any overflow in one or two "iterations."

As an illustration of this method of overflow detection, let us reconsider the second example of multiplication given above. We have $M/2 = 105$, so we set $k = 10$ and $d = 5$. Also, $x_0 = 29$, $y = 9$, and $z_0 = 51$. Since $z_0 > |x_0| < |y|$, it is necessary to calculate $a_0 = |x_0| - k = 19$ and $b = |y| - k = -1$. Then $a_0 = 19 > |b| = 1$, so we set $x_1 = a_0 + b = 18$. Now, $x_1 |y| = 18 \cdot 9 = 162 \equiv -48$ (mod 210), which gives $z_1 = -48 < |x_1|$. This indicates overflow has occurred.

4.1.2.3 Multiplicative Overflow - Method II

Since the overflow detection method given in the preceding paragraph requires so many operations, we have devised a simpler method which makes use of a stored table. This method, unlike the preceding one, is valid only if "two-sided" residues are used or if integers between M/2 and M are regarded as being negative (that is, any x such that $M/2 < x < M$ would represent the negative integer, $x - M$). Such provision would probably be made for negative integers in a modular arithmetic computer.

As in Method I, we first compare $z \equiv |x| \cdot |y|$ (mod M) with $|x|$ and $|y|$. If $|z| > |x| \geq |y|$, we refer to a table consisting of all integer powers of 2 from $2^0 = 1$ through 2^n, where $2^n \leq M/2 < 2^{n+1}$, to find integers, p and q, such that $2^{p-1} < |x| \leq 2^p$ and $2^{q-1} < |y| \leq 2^q$. If $p + q \leq n$, then $|x| \cdot |y| \leq 2^{p+q} \leq 2^n$, and no overflow occurs. Similarly, if $n + 3 \leq p + q$, then $2^{n+1} \leq 2^{p+q-2} < |x| \cdot |y|$, and overflow occurs.

If $p + q = n + 1$, then $2^{n-1} < |x| \cdot |y| \leq 2^{n+1} \leq M$, and any overflow will have been detected by the presence of the "wrong" sign on z. Moreover, if $p + q = n + 2$, then $2^n < |x| \cdot |y| \leq 2^{n+2} \leq 2M$, and any overflow such that $M/2 < |x| \cdot |y| \leq M$ or $3M/2 < |x| \cdot |y| \leq 2M$ will also have been detected by the "wrong" sign on z.

We must now differentiate between the two following cases for $p + q = n + 2$:

Case 1 - $2^n < |x| \cdot |y| \leq M/2$, and overflow does not occur;

Case 2 - $M < |x| \cdot |y| \leq 3M/2$, and overflow does occur. To do this, we multiply 2^{p-1} by $|y|$ and examine the sign of the result. In Case 1 we have $2^{n-1} < |x/2| \cdot |y| \leq 2^{p-1} |y| < |x| \cdot |y| \leq M/2$, so the sign of $z_1 \equiv 2^{p-1} |y|$ (mod M) will be plus; but in Case 2, we have $M/2 < |x/2| \cdot |y| \leq 2^{p-1} \cdot |y| \leq 2^{p-1} \cdot 2^q = 2^{n+1} \leq M$, so the sign of z_1 will be minus.

For example, the overflow in the multiplication of 9 by 29 in paragraph 4.1.2.2 above is detected by Method II as follows: $x = 29$, $y = 9$, and $x = 51$, so $|z| > |x| > |y|$. We refer to the table to find $p = 5$ and $q = 4$. Then, $p + q = 9 > 8 = n + 2$, which indicates overflow.

As a second example, consider the multiplication of $x = -31$ and $y = 8$ in the same system. Then $x \cdot y = -248 \equiv -38$ (mod 210), and $|z| > |x| > |y|$. From the table we find $p = 5$ and $q = 3$. This gives $p + q = 8 = n + 2$, so we calculate $z_1 = 2^{p-1} |y| = 16 \cdot 8 = 128 \equiv -82$ (mod 210). Since the sign of z_1 is minus, overflow has occurred.

We can also give a different treatment to the above situation where $n + 1 \leq p + q \leq n + 2$. If we define a and b by $a = |x| - 2^{p-1}$ and $b = |y| - 2^{q-1}$, then

$$|x| \cdot |y| = 2^{p+q-2} + a \cdot 2^{q-1} + b \cdot 2^{p-1} + ab. \tag{2}$$

Since $0 < a \leq 2^{p-1}$ and $0 < b \leq 2^{q-1}$, then each of the four terms on the right side of equation 2 is $\leq 2^{p+q-2} \leq 2^n$. Hence, multiplicative overflow occurs in $|x| \cdot |y|$ if and only if additive overflow occurs in equation 2.

For example, overflow can be detected in the above multiplication of $x = -31$ and $y = 8$ as follows:

From equation 2 we have

$$|x| \cdot |y| = 2^6 + 15 \cdot 2^2 + 4 \cdot 2^4 + 15 \cdot 4 = 64 + 60 + 64 + 60.$$

Overflow occurs in the addition of any two of these terms, so $|x| \cdot |y|$ overflows.

In general, this treatment of the case where $n + 1 \leq p + q \leq n + 2$ requires more operations than the treatment involving $z_1 = 2^{p-1} |y|$, since two to six magnitude comparisons are required to detect the presence or absence of additive overflow in equation 2. Therefore, despite the appealing conceptual simplicity of the treatment using equation 2, the other is preferable since it requires less machine operations.

The requirement of the table of powers of 2 for Method II is really quite modest even for modular arithmetic computers with a large machine range (for example, if $M = 10^9$, $n = 29$), and as we will show in paragraph 4.1.5, the table can be used for other important modular arithmetic operations.

While the overflow detection methods given above are not as simple as those commonly used in most binary computers, they are the simplest yet obtained for modular arithmetic. They should not be "wired into" a computer and performed automatically after every addition and multiplication, but rather should be used where needed in carefully constructed programs.

4.1.2.4 "Problem-Oriented" Overflow Control

In many of the most important computations that occur (for example, the tabulation of a polynomial or the iterative solution of an algebraic or ordinary differential equation), we must calculate a sequence of integers which satisfy a certain system of recurrence relations. Oftentimes, these recurrence relations can be used to detect overflow in the calculation of an element of the sequence from its predecessors.

For instance, if $x_1, x_2, \ldots,$ is a non-decreasing sequence of integers such that $x_n \leq ax_{n-1}$ for some positive integer a and for all n, then the integer a can be used as a redundant "checking modulus" to detect overflow, provided that a is less than M, the product of the moduli, and is relatively prime to all the moduli. That is, if the residues $a_i \equiv x_i \pmod{a}$ are calculated along with the "usual" residues, then a_i can be used to detect overflow in the form $M \leq x_i < aM$, where $x_{i-1} < M$. We calculate a_i', the trial residue of x_i modulo a, from the residues of x_i relative to the usual moduli and compare it with the "true" residue of $x_i \pmod{a}$ which is a_i. If $a_i' \not\equiv a_i$, then overflow has occurred in the calculation of x_i from x_{i-1}.

As an extremely simple example, let us use the moduli 2, 3, 5, and 7, and assume the recurrence relation to be $x_n = 10x_{n-1} - x_{n-2}$, where $x_0 = x_1 = 1$. Then, since $x_{n-2} > 0$, it follows that $x_n < 10x_{n-1} < 11x_{n-1}$. Hence, we can use $a = 11$ as "checking modulus" because $11 < 2 \cdot 3 \cdot 5 \cdot 7 = 210 = M$ and has no common divisors with any of the moduli. The calculation proceeds as follows:

n =	0	1	2	3	4	5
x_n =	1	1	9	89	881	8721
a_n =	1	1	-2	1	1	-2
a_n' =	1	1	-2	1	-3	1

Since a_4 and a_4' are unequal, we conclude that overflow occurred in the calculation of x_4 Furthermore, since $a_4 - a_4' = 4$ and $M \equiv 1 \pmod a$, we can solve the congruence $z \equiv 4 = a_4 - a_4' \pmod a$ to yield $z \equiv 4$; from this we conclude that 4 "overflows" have occurred - that is, $4M \leq x_4 < 5M$.

It should be stressed that, when using this method of overflow detection, it is essential to notice the first discrepancy which occurs between a_i and a_i', for later a_i and a_i' may spuriously agree. Also, the estimate of the amount of "overflow" is valid only for the first x_i which overflows.

The principal utility of this technique, which is concerned only with overflow of the final result of a calculation, is in complicated formulas in which overflow of individual terms may be cancelled by other terms of the formula. An example of such a formula is the error estimate, e_i, of the division process described in a later paragraph. (However, in that case, it is proven that e_i never overflows, so that no detection is required.) It may also be noted that this same philosophy of "problem-oriented" overflow control may in some cases be applied without the use of redundant moduli by comparing x_{n-1} with $\left[M/a\right]$ as is done in paragraph 4.2.2.

4.1.3 Division by a Modulus or a Product of Moduli-Scaling and Round-Off

We are given the number x in modular form, $x = (x_1, x_2, \ldots, x_n)$, in the system with moduli $m_1, m_2, \ldots, m_n$ where the x_i's are either least nonnegative or least absolute value residues. We wish to divide x by one of the moduli, say m_i. In general, x will not be divisible by m_i, although $(x - x_i)$ is; therefore, the integer result which the process will give is generally in error by x_i/m_i. There is no problem in obtaining the residues, x_j^*, of the quotient $x^* = (x - x_i)/m_i$ from the congruences

$$m_i x_j^* \equiv (x_j - x_i) \pmod{m_j}, \quad j = 1, 2, \ldots, n, \qquad j \neq i \tag{3}$$

since if $j \neq i$, the moduli are relatively prime. Only the residue, x_i^*, presents difficulties. If $j = i$, equation 3 does not have a unique solution, and another technique must be used to find this particular residue.

The residue, x_i^*, can be computed by using the property that, in the mixed radix representation of x^* in which m_i is taken as the last modulus, the last coefficient must be zero. To see this, consider the mixed radix representation of x with m_i taken first:

$$x = x_i + a_1 m_i + a_2 m_1 m_i + \ldots + a_{n-1} m_1 m_2 \ldots m_i \ldots m_{n-1}. \qquad (4)$$

Then,

$$x^* = \frac{x - x_i}{m_i} = a_1 + a_2 m_1 + \ldots + a_{n-1} \frac{m_1 m_2 \ldots m_i \ldots m_{n-1}}{m_i}. \qquad (5)$$

Comparing equation 5 with the mixed radix representation of x^* in which m_i is taken last gives

$$x^* = b_1 + b_2 m_1 + \ldots + b_{n-1} \frac{m_1 m_2 \ldots m_{n-1}}{m_i} + b_n \frac{m_1 m_2 \ldots m_n}{m_i}. \qquad (6)$$

This comparison shows that

$$b_1 = a_1,\ b_2 = a_2,\ \ldots,\ b_{n-1} = a_{n-1},\ \text{but } b_n = 0. \qquad (7)$$

(Alternatively, $b_n = 0$ follows from the fact that the last coefficient in the mixed radix representation provides an estimate of the magnitude of a number. For example, with least nonnegative residues, $b_n\ M/m_i \leq x^*$.
But,

$$x^* = \frac{x - x_i}{m_i} < \frac{M - x_i}{m_i} < \frac{M}{m_i}, \qquad (8)$$

where M is the product of all the moduli. Hence, $b_n = 0$.) The reduction of x^* to the mixed radix representation shown in equation 6 leads to a congruence, by the technique of paragraph 4.1.1,

$$C_1 x_i^* - C_2 \quad b_n \quad 0 \pmod{m_i}, \qquad (9)$$

for the unknown residue, where

$$C_1 = \prod_{j \neq i} d_{ji} \pmod{m_i} \qquad (10)$$

and C_2 is determined by the known x_j^*'s. The constant, C_1, or, better yet, its inverse which is simply $\prod m_j \pmod{m_i}$ may be precomputed and stored. Note that C_2 is the last mixed radix coefficient of the modular number:

$$(x_1^*, x_2^*, \ldots, x_{i-1}^*, 0, x_{i+1}^*, \ldots, x_n^*), \tag{11}$$

so that x_i^* is the product (modulo m_i) of this coefficient and the inverse of C_1.

To divide x by the product of several moduli, say $m_1 m_2 m_3$, we proceed in a similar fashion. (Any of the moduli may be chosen; these particular ones are used here for convenience of notation.) In general, x will not be divisible by $m_1 m_2 m_3$, but $x - x_{1,2,3}$ is, where $x_{1,2,3}$ is its residue mod $m_1 m_2 m_3$. Therefore, the integer result $(x - x_{1,2,3})/m_1 m_2 m_3$ is in error by no more than $x_{1,2,3}/m_1 m_2 m_3$. We obtain $x_{1,2,3}$ by performing the first few steps of the mixed radix conversion of x with m_1, m_2, and m_3 as the first three moduli:

$$x = c_1 + c_2 m_1 + c_3 m_1 m_2 + c_4 m_1 m_2 m_3 + \ldots + c_n \overline{\prod_{i=1}^{n-1}} m_i \tag{12}$$

Subtracting $b = c_1 + c_2 m_1 + c_3 m_1 m_2$ from both sides of equation 12 and dividing by $m_1 m_2 m_3$ results in

$$x^* = \frac{x - b}{m_1 m_2 m_3} = c_4 + c_5 m_4 + \ldots c_n \overline{\prod_{i=4}^{n-1}} m_i \tag{13}$$

The magnitude of b must be investigated. For least nonnegative residues

$$b \leq m_1 m_2 m_3 - 1 \tag{14}$$

and hence b is $x_{1,2,3}$. For least absolute value residues, extra care must be taken. If an even modulus is among m_1, m_2, and m_3, it must be eliminated first, so that b is $x_{1,2,3}$. Each residue, x_i^*, of x^* of the form

$$x_i^* \equiv \frac{x_i - c_1}{m_1 m_2 m_3} - \frac{c_2}{m_2 m_3} - \frac{c_3}{m_3} \pmod{m_i} \tag{15}$$

for $i \neq 1, 2, 3$ is computed mod m_i. The residues x_1^*, x_2^*, and x_3^* are indeterminate as before, but are found by a process analogous to that used in the division by a single modulus. In the equations analogous to equation 7, the last three coefficients can be shown to be 0. Hence, we proceed with the mixed radix conversion in which m_1, m_2, and m_3 are the last to be eliminated, until we come to a point where we have three terms in the form of equation 9. At this point, each of these terms can be set congruent to 0. The constants analogous to $(C_1)^{-1}$ of equation 9 are precomputable as before; and in fact are

$\prod_{j>3} m_j \pmod{m_i}$ $i = 1, 2, 3;$

and x_i^* is found as before.

A clarification of the size of the errors involved follows. Just after equation 14, mention was made of one peculiar effect the even modulus can have in a least-absolute-value residue system. In this same system, the residues of the even modulus, m, differ in one basic aspect from those of the odd moduli in that one of the residues is equal to m/2 in absolute value. The choice of whether it is positive or negative determines whether a quotient with a fractional part 1/2 is rounded up or down. The positive choice leads to rounding down, the negative to rounding up. Furthermore, some care must be taken to ensure that in the rounding process, a number very near the open end of the range (either + M/2 or -M/2) is not rounded in such a way as to cause overflow.

For example, if we let $m_1 = 2$, $m_2 = 3$, $m_3 = 5$, and $m_4 = 7$, we can divide $93 \leftrightarrow (1, 0, -2, 2)$ by $m_2 m_3 = 15 \leftrightarrow (1, 0, 0, 1)$ as follows:

To find the residue of 93 (mod 15), we perform the first two lines of the two-sided mixed radix conversion of 93, taking the moduli m_2 and m_3 first.

Modulus	3	5	2	7
Residue	$\underline{0}$	-2	1	2
		$\underline{1}$	1	3

Hence, $93 \equiv 0 + 1\,(3) = 3 \pmod{15}$. Next, we subtract the residue of 93 (mod 15) from 93 to obtain $90 \leftrightarrow (0, 0, 0, -1)$, and solve the simultaneous linear congruences corresponding to $15\,x^* = 90$: $(1, 0, 0, 1) \cdot (x_1^*, x_2^*, x_3^*, x_4^*) = (0, 0, 0, -1)$. The results are $x_1^* = 0$, $x_4^* = -1$, and x_2^*, x_3^* are indeterminate. We set $x_2' = x_3' = 0$, and expand $(x_1^*, x_2', x_3', x_4^*)$ in two-sided mixed radix form, taking m_2 and m_3 last.

Modulus	2	7	3	5
Residue	$\underline{0}$	-1	0	0
		$\underline{3}$	0	0
			$\underline{0}$	-1

Using the elements from the last row of the above array, we have $x_2^* = m_1 m_4 (0) \equiv 0 \pmod 3$ and $x_3^* = m_1 m_4 (-1) \equiv 1 \pmod 5$. Hence, the "nearest integer" to 93/15 is $6 \leftrightarrow (0, 0, 1, -1)$.

4.1.4 Conversion to Fixed Radix Notation

Let the binary expansion of a number w be:

$$w = 2^k c_o + 2^{k-1} c_1 + \ldots 2c_{k-1} + c_k.$$

Define

$$w_i = 2w_{i-1} + c_i$$

and

$$w_k = w.$$

Then clearly

$$w_i \equiv c_i \pmod 2$$

and

$$w_{i-1} = (w_i - c_i)/2, \text{ an integer.}$$

The purely modular method for performing this division to determine w_{i-1} in modular form has been described in paragraph 4.1.3 above. Thus starting with $w_k = w$, the successive bits of w are obtained in the order of increasing significance as the residues mod 2 of the w_i.

The conversion is made as readily to any radix which is one of the prime moduli of the system. Composite radices such as 10 may also be utilized if their prime factors (2 and 5) are used as moduli. In this case, the division procedure of paragraph 4.1.3 yields successively the residues mod 2 and mod 5 of the digits of x, a sort of modular coded decimal notation. The conversion to ordinary decimal is quite simple - the digit is equal to its residue mod 5 when this residue is congruent to the residue of the digit mod 2; if not, the digit is equal to 5 plus its residue mod 5.

In conversion to decimal form, care must be taken to ensure that positive digits are computed. For conversion to binary, it is only necessary that 0, + 1 be used as the residues mod 2.

The advantage of this conversion technique is that no special hardware is required. However, it is shown in paragraph 6.3.1 that for a machine using n moduli with a decimal range of d digits, this procedure requires about 2d (n + 4) +3 clock times as compared with n clock times for the high speed implementation of the Chinese Remainder Theorem described in paragraph 6.2.5. Furthermore, the least significant digits are generated first in the above procedure.

4.1.5 Division

The problem of division in MA is of great importance since many significant problems require division at some point in their computer solution. Several approaches to division have been tried with varying degrees of success and they are described below, together with the particular difficulties arising in their use.

4.1.5.1 Approximation of Divisor

A method for approximating the quotient of two integers by another integer has been suggested in reference 1. This method consists of choosing some product of moduli as an approximation for the divisor and applying a suitable algorithm to obtain either the greatest integer less than or equal to the quotient, or the integer nearest to the quotient (that is, the quotient "rounded-off" to the nearest integer). If we wish to approximate $x/y = q$ by $\{q\}$, the integer nearest to q, the algorithm is given by defining

$$q_o = \{x/z\}, \; r_o = x - q_o y \tag{16}$$

$$q_i = q_{i-1} + \{r_{i-1}/z\}, \; r_i = r_{i-1} - \{r_{i-1}/z\},$$

where z is a product of moduli such that $y \leq z < 2y$. The integer approximations of the divisions by z are obtained by the method described in paragraph 4.1.3 above. From the definitions it may be readily shown that $r_i = r_{i-1}$ if and only if $r_{i-1} < \frac{1}{2} z$ and that $r_i \leq r_{i-1} - 1$ if and only if $r_i < \frac{1}{2} z$. (What happens when $r_{i-1} = \frac{1}{2} z$ depends on whether $\frac{1}{2}$ is rounded up or down in the procedure given in paragraph 4.1.3) Thus, the algorithm stops automatically whenever the best q_i has been found. However, since z may be greater than y, a final check must be performed. If $2 |r_{i-1}| > y$ (that is, $(|r_{i-1}| > \frac{1}{2} y)$), q_i must be increased or decreased by one, according as r_{i-1} is positive or negative, to give the integer $\{q\}$ closest to x/y. Or, if it is desired to find the greatest integer less than x/y (that is, $[x/y]$), q_i must be decreased by one if $r_{i-1} < 0$.

A somewhat detailed analysis gives the result that a sufficient condition for $|r_{i-1}| < \frac{1}{2} z$ is that $i \geq - \log (x - z)/\log t$, where $y/z = 1 - t$. However, this condition is not necessary, although it does provide a reasonably good estimate of the number of iterations involved for certain choices of y. Certain other variations of the above algorithm are possible, but the one stated above seems to be the most efficient in terms of computation required.

While the above procedure requires only purely modular operations, it presents some difficulties in its utilization. First, it seems that z would have to be determined by means of a table look-up procedure. The usual requirement that the moduli be relatively prime combined with the requirement that there always exist a z between y and 2y impose restrictions on the choice of moduli so that it seems the table would necessarily consist of all products of the first n primes (they being the n moduli) taken 1, 2, ..., n - 1, at a time. Such a table would then contain $(2^n - 2)$ words, a rather formidable number for all but small n Secondly, the above algorithm is capable of giving only integer approximations.

4.1.5.2 Newton-Raphson and Similar Methods

A Newton-Raphson method for computing x/y by an iterative procedure has been suggested in reference 1, pp 93-97, and another similar iterative procedure was suggested in reference 3. These methods seem adaptable to MA if the denominators are kept in a certain form, say, powers of 2. While these methods do converge fairly rapidly to a good approximation of x/y, overflow is usually encountered after only 2 or 3 iterations - long before the desired accuracy of approximation is obtained. This overflow would not be a problem if some means of reducing numerator and denominator by a common factor were known, but no convenient method of doing this has been found and a method which would retain the proper form of the denominator seems particularly difficult. It should be mentioned that a multiple-word method of representing integers in MA using M as a radix has been suggested in reference 1, chapters 8 and 9, as well as in reference 2, and in the former report, the Newton-Raphson method was used for the division of numbers represented in this way. However, in addition to requiring two to three times as much storage capacity, this representation of numbers requires such complicated addition, multiplication, and sign determination methods that all the inherent advantages of MA may be lost in its use.

4.1.5.3 Division Using Powers of Two

A somewhat efficient division procedure which uses a stored table of powers of two has been described in Y.A. Keir, P.W. Cheney, and M. Tannenbaum, "Division and Overflow Detection in Residue Number Systems," IRE Transactions on Electronic Computers, Vol. 1 EC-11, August, 1962, pp. 501 - 507. This method, which is essentially a variation of the usual method of division by repeated subtraction of the divisor from the dividend, generates exactly one binary bit of the quotient per iteration. It is not subject to overflow error and seems to require little "extra hardware," but it gives only the "integer part" of the quotient and is not as fast as might be desired.

4.1.5.4 An Optimal Division Method

We shall now describe another method which uses a stored table of powers of two. However, instead of giving only the "integer part" of the quotient as does the method described immediately above, the following method approximates the quotient, a/b, of two integers by a "fraction" in the form $x \cdot 2^j$, where x is an integer in residue form and j is a nonnegative integer. The convergence in this method is nearly three times as rapid as that of the method described in paragraph 4.1.5.3.

We assume that a and b are nonzero and that their absolute values are both less than or equal to M/2, where M is the product of the moduli. Furthermore, we assume the availability of a stored table of powers of 2 from 2^0 to 2^n, where n is the integer such that $2^n \leq M/2 < 2^{n+1}$. Each element of this table will consist of the "two-sided" mixed radix coefficients of a power of 2, the residues of the same number, and the corresponding exponent of 2.

To obtain an initial approximation, x_1, to $|a/b|$, we convert a and b to "two-sided" mixed radix notation, and change signs, if necessary, to obtain the absolute values of a and b (at the same time, we can also determine the sign of the quotient from the signs of a and b). Next we use the "two-sided" mixed radix coefficients of $|a|$ and $|b|$ to find integers p and q in the table such that

$$2^{p-1} < |a| \leq 2^p \tag{17}$$

and

$$2^{q-1} < |b| \leq 2^q.$$

It follows that

$$2^{p-q-1} = 2^{p-1}/2^q < |a/b| < 2^p/2^{q-1} = 2^{p-q+1}; \tag{18}$$

therefore, we set the first approximation $x_1 = 2^{p-q-j}$, where j is determined as described below.

If we define e_1 by

$$e_1 = 2^{-j} \quad |a| - |b| \cdot x_1,$$

it follows from equations 17 and 18 that

$$(2^{-j} \cdot 2^{p-1} - 2^q \cdot 2^{p-q-j}) < e_1 < (2^{-j} \cdot 2^p - 2^{q-1} \cdot 2^{p-q-j}), \tag{19}$$

which gives

$$|e_1| < 2^{p - j - 1}$$

Since $|a| \le M/2$, it follows that $p \le n + 1$, so e_1 will not overflow if j is chosen such that $0 \ge j \ge (p - n - 1)$.

Since, by definition, $e_1 = 2^{-j} \cdot b \cdot (a/b - x_1 \cdot 2^j)$, e_1 is a convenient estimate the error in the approximation of a/b by $x_1 \cdot 2^j$. To obtain a more precise estimate of this error, we convert e_1 to "two-sided" mixed radix form, determine its sign, and refer to the table to find an integer z_1 such that $2^{z_1 - 1} < |e_1| \le 2^{z_1}$. Next, we improve our approximation to $|a/b|$ by defining x_2 by $x_2 = x_1 \pm 2^{z_1 - q}$, taking the plus sign if e_1 was positive, and the minus sign if e_1 was negative. Obviously, we can continue this process, defining e_i, z_i, and x_i by:

$$e_i = 2^{-j} \cdot |a| - |b| \cdot x_i ; \; 2^{z_i - 1} < |e_i| \le 2^{z_i} ; \tag{20}$$

and

$$x_{i+1} = x_i \pm 2^{z_i - q} ,$$

where the plus or minus sign is used according as $e_i > 0$ or $e_i < 0$.

From the definitions (equation 20) of e_i and x_i, it follows that $|e_{i+1}| = \pm\left(|e_i| - 2^{z_i - q} \, |b|\right)$ If the plus sign holds, we may apply equations 17, 18, and 20 to yield:

$$|e_{i+1}| < \left(|e_i| - 2^{z_i - q} \cdot 2^{q-1}\right) \le \left(2^{z_i} - 2^{z_i - 1}\right) = 2^{z_i - 1} ; \tag{21}$$

similarly, if the minus sign holds, then

$$|e_{i+1}| \le \left(2^{z_i - q} - 2^q - |e_i|\right) < \left(2^{z_i} - 2^{z_i - 1}\right) = 2^{z_i - 1} . \tag{22}$$

Hence $z_{i+1} \le z_i - 1$, which implies that the sequence, $x_1, x_2, \ldots,$ converges to $|a/b|$ with an increase in accuracy of at least one binary "bit" per "iteration."

If desired, j can be decreased in succeeding "iterations." If $0 \ge j \ge (z_i - n - 1)$, it follows from the above analysis that $|e_{i+1}| \le 2^n \le M/2$. In addition, it follows from equation 18 that if $j \ge (p - 2 - n + 1)$, then all x_i are less than or equal to 2^n

Hence, by keeping j within these bounds, overflow in e_i and x_i can be prevented. (It should be pointed out that each of the terms $2^{-j} \cdot |a|$ and $|b| \cdot x_i$ will often overflow considerably, but because of the unique characteristics of the modular arithmetic number system, the difference of these terms, which is e_i, will be correct and will be within machine range if $0 \geq j \geq (z_{i-1} - n - 1.)$

If the sequence, $x_1, x_2, \ldots,$ is terminated with, say, x_s, a final test may be necessary to ensure maximum accuracy. If $z_s = q - 1$, we compare $|2e_s|$ with $|b|$. If $|2e_s| \geq |b|$, we increase or decrease x_s by 1 according as $e_s > 0$ or $e_s < 0$; otherwise, we leave x_s unchanged. This guarantees that

$$e_s = 2^{-j} \cdot |b| \cdot | \, |a/b| - x_s | \leq |b|/2, \tag{23}$$

from which it follows that $|a/b - x_s \cdot 2^j| = |e_s| \cdot |b| . \;\; 2^j \leq 2^{j-1}$. If $z_s < q - 1$, this test is unnecessary since $|a/b - x_s \cdot 2^j| = |e_s| \cdot |b| \cdot \;\; 2^j < 2^{z_s} \cdot 2^{q-1} \cdot 2^j = 2^{z_s + q - 1 + j} \leq 2^{j-1}$.

To illustrate the unusual flexibility of the above division method, we now present two variations of that method. In the first, we have adapted the method to give maximum efficiency in finding the integer nearest to the quotient. In the second, the method is adapted so as to give "optimal" performance; that is, j is controlled so as to give the maximum possible $|e_i|$'s and x_i, which ensure, respectively, maximal speed of convergence and maximum accuracy obtainable with the given machine range. A third variation, not described here, permits the modular arithmetic computer programmer to control the accuracy in each division by specifying a $j \leq 0$ in the division instruction.

For clarity, we present these two variations in step-by-step form, followed by illustrative examples.

a. Variation 1 - Nearest Integer Division

Step 1 - Convert a and b to "two-sided" mixed radix form and determine their signs. If b = 0, turn on the divide error indicator (if any) and halt. Otherwise, change the signs of the mixed radix coefficients for a and b, if necessary, so as to obtain $|a|$ and $|b|$. At the same time, set the sign of x to be plus if and only if a and b have the same sign. Compare $|a|$ and $|b|$; if $|a| \leq |b|$, go to Step 2; otherwise, go to Step .

Step 2 - Calculate $2|a|$, convert it to "two-sided" mixed radix form, and determine sign. Compare $2|a|$ and $|b|$. If $2|a| \geq |b|$ or $2|a| \leq 0$, set the magnitude of x equal to 1; otherwise, set x = 0. Terminate division.

Step 3 - Refer to the table to find integers p and q satisfying equation 17 above. Calculate $r = p - q$ and obtain 2^r in residue form from the table. Set x', the magnitude of x (without sign), equal to 2^r and go to Step 4.

Step 4 - Calculate $e = |a| - |b| \cdot x'$, convert it to "two-sided" mixed radix form, and determine its sign. If $e = 0$, terminate division, otherwise, go to Step 5.

Step 5 - If necessary, change the sign of the "two-sided" mixed radix coefficients for e to obtain $|e|$. Refer to the table to find an integer z such that $2^{z-1} < |e| \leq 2^z$. Compare z and $q - 1$. If $z > q - 1$, go to Step 6; if $z < q - 1$, terminate division; otherwise go to Step 7.

Step 6 - Calculate $t = z - q$. Refer to the table to find 2^t in residue form and increase or decrease x' by that amount according as $e > 0$ or $e < 0$. Go back to Step 4.

Step 7 - Calculate 2e, convert it to "two-sided" mixed radix form, and determine its sign. Change signs, if necessary, to obtain $|2e|$. Compare $|2e|$ and $|b|$. If $|2e| \geq |b|$, increase or decrease x' by 1 according as $2e > 0$ or $2e < 0$; otherwise, leave x' unchanged. Terminate division.

b. Variation 1 - Example

Let $a = 136{,}047$ and $b = 85$.

Then,

$a/b = 1600.5529 \ldots.$

Since $|a| > |b|$, we execute Step 3 and obtain $p = 18$, $q = 7$. Let us now refer to each "cycle" through Steps 4-7 as an "iteration." The results of the successive iterations are:

Iteration 1	Iteration 2	Iteration 3
$x' = 2^{11} = 2048$	$x' = 2048 - 2^9 = 1536$	$x' = 1536 + 2^6 = 1600$
$e = -38{,}033$	$e = 5487$	$e = 47$
$z = 16$	$z = 13$	$z = 6 = q - 1$

At the end of Iteration 3, Step 7 is executed with the result that $|2e| = 94 > 85 = |b|$. Therefore, x' is increased by 1 to give the final result $x = 1601$.

c. Variation 2 - "Optimal" Division

Step 1 - Convert a and b to "two-sided" mixed radix form and determine their signs. If $b = 0$, turn on the divide error indicator and halt. Otherwise, obtain $|a|$ and $|b|$ and set the proper sign on x. Set $j = 0$. If $a = 0$, set $x = 0$ and terminate division; otherwise go to Step 2.

Step 2 - Refer to the table to find integers p and q satisfying equation 17 above. Calculate $r = p - q$ and obtain 2^r in residue form from the table. Set x', the magnitude of x, equal to 2^r. Calculate $e = |a| - |b| \cdot x'$, convert it to "two-sided" mixed radix form, and determine its sign. If $e > 0$, set $j_{min} = (p - q - n + 1)$; otherwise, set $j_{min} = (p - q - n)$. Go to Step 3a.

Step 3 - Calculate $e = 2^{-j} \cdot |a| - |b| \cdot x'$, convert it to "two-sided" mixed radix form, and determine its sign.'

Step 3a - If $e = 0$, replace x' by $x' \cdot 2^{(j - j_{min})}$, set $j = j_{min}$, and terminate division; otherwise, go to Step 4.

Step 4 - Obtain $|e|$ and refer to the table to find an integer z such that $2^{z - 1} < |e| \leq 2^z$. Calculate $t = z - q + j - j_{min} + 1$. If $t > 0$, go to Step 5; if $t = 0$, go to Step 6; otherwise, replace x' by $x' \cdot 2^{(j - j_{min})}$, set $j = j_{min}$, and terminate division.

Step 5 - Calculate $u = z - n - 1$, and set $v = \text{Max}(j + u, j_{min})$. Obtain the residue form of $2^{j - v}$ from the table, and replace x' by $2^{j - v} \cdot (x' \pm 2^{z - q})$, taking the plus or minus sign according as $e > 0$ or $e < 0$. Replace j by v and go back to Step 3.

Step 6 - Replace x' by $x' \cdot 2^{(j - j_{min})}$ and set $j = j_{min}$ Calculate $e' = e \cdot 2^{(j - j_{min} + 1)}$, convert it to "two-sided" mixed radix form, determine its sign, and obtain $|e'|$. Compare $|e'|$ and $|b|$. If $|e'| \geq |b|$, increase or decrease x' by 1 according as $e' > 0$ or $e' < 0$; otherwise leave x' unchanged. Terminate division.

Note: If division is terminated with $t = 0$, the error is not greater than $2^{j_{min} - 1}$; if $t < 0$, it can be shown that the error is strictly less than $2^{t + j_{min}} \leq 2^{j_{min} - 1}$. If $t = +1$, the approximation is "exact"; i.e., the error is zero.

d. Variation 2 - Example

Unlike Variation 1, Variation 2 requires that the machine range be specified because n, the integer such that $2^n \leq M/2 < 2^{n + 1}$, is used. Hence, we assume the moduli to be the eight primes from 2 through 19, which gives a machine range of $M/2 = 4{,}849{,}845$. Since $2^{22} = 4{,}194{,}304$; $n = 22$. Let $a = 829{,}314$ and $b = 6{,}057$. Then $a/b = 136.91827637\ldots$

Step 2 gives $p = 20$, $q = 13$, and $j_{min} = -14$. As in the example for Variation 1, we give the results of the successive "iterations" below:

Iteration 1

$x' = 2^7 = 128$

$j = 0$

$e = 54,018$

$z = 16$

$t = 18$

Iteration 2

$x' = 2^7 \quad (128 + 2^3) = 17,408$

$j = -7$

$e = 711,936$

$z = 20$

$t = 15$

Iteration 3

$x' = 2^3 \quad (17.408 + 2^7)$

$= 140,288$

$j = -10$

$e = -506,880$

$z = 19$

$t = 11$

Iteration 4

$x' = 2^4 \cdot (140,288 - 2^6)$

$= 2,243,548$

$j = -14$

$e = -1,907,712$

$z = 21$

$t = 9$

Iteration 5

$x' = 2,243,584 - 2^8$

$= 2,243,328$

$j = -14$

$e = -357,120$

$z = 19$

$t = 7$

Iteration 6

$x' = 2,243,328 - 2^6$

$= 2,243,264$

$j = -14$

$e = 30,528$

$z = 15$

$t = 3$

Iteration 7

$x' = 2,243,268 + 2^2$

$= 2,243,268$

$j = -14$

$e = 6300$

$z = 13$

$t = 1$

Iteration 8

$x' = 2,243,268 + 2^0$

$= 2,243,269$

$j = -14$

$e = 243$

$z = 8$

$t = -4$

Division is terminated at this point, with a/b being approximated by $2,243,269 \cdot 2^{-14}$. It should be pointed out that while x' is best for $j = -14$, less than "optimal" accuracy was obtained for this example since $2^n < 2x < M/2$ implies that j could have been -15. To minimize occurrences such as this, the moduli should be chosen so as to make M/2 approximate from below some power of 2.

The following table shows the convergence in the above example.

a/b = 136.91827637...

Iteration	x'	j	$x \cdot 2^j$	Error
1	128	0	128.00000000	-8.91827637...
2	17,408	-7	136.00000000	-0.91827637....
3	140,288	-10	137.00000000	+0.08172363...
4	2,243,584	-14	136.93750000	+0.01922363...
5	2,243,328	-14	136.92187500	+0.00359863...
6	2,243,264	-14	136.91796875	-0.00030762...
7	2,243,268	-14	136.91821289	-0.00006348...
8	2,243,269	-14	136.91827392...	-0.00000245...

The maximum error predicted is less than $2^{t + j_{min}} = 2^{-16} = 0.00001526....$

From the above examples, the advantages of this division method are obvious:

a. No overflow is possible so long as j is kept within the bounds prescribed above.

b. The convergence is quite rapid; we have shown that at least one binary bit of accuracy must be obtained per iteration, and in practice the convergence is usually much faster than that. For example, in several simulator tests, each of which involved 500 "randomly chosen" "optimal divisions," an average accuracy of 2.85 bits per iteration was obtained.

c. Any desired degree of accuracy, from the nearest integer to the maximum possible for the machine range, can be attained.

Furthermore, the additional circuitry required to implement this method in modular arithmetic computers is modest. The table of powers of 2 is quite small, considering the machine range, and it can also be used for overflow detection. Additional hardware is necessary to handle j, but we anticipate that small binary adders and registers will suffice.

It should be mentioned that the success of the above division method is dependent upon the fact that powers of two are used. A brief investigation was conducted to see if some other "radix" such as 8 or 10 could be used, but the results indicated that enough complications would arise so as to render the method highly impractical if not impossible.

4.1.6 Square Roots

The usual method for extracting square roots on a digital computer is the Newton-Raphson method, in which the successive approximations, w_i, to $\sqrt{a^2}$ are generated by the equation,

$$w_{i+1} = \frac{1}{2}(w_i + a^2/w_i). \tag{24}$$

Although for any positive initial approximation, w_0, the sequence, $w_1, w_2, w_3, \ldots,$ converges quite rapidly to the positive square root of a^2, the method is not well suited for modular arithmetic computation because at least one division is required to obtain each w_{i+1} from w_i. Even with the "optimal" division procedure described in the preceding paragraph, the amount of computation required is large enough to offset the advantage of rapid convergence, particularly, when it becomes necessary to use "floating-point" arithmetic (see paragraph 4.1.7) to avoid the introduction of errors by rounding-off to the nearest integer. Therefore, we have sought to modify the "optimal" division procedure itself to enable us to extract square roots directly. The resulting procedure, which is described below, does not converge as rapidly per iteration as the Newton-Raphson method, but because it involves much less computation per term, the following method requires considerably less total computation for any given degree of accuracy.

Given a^2 in residue form and such that $0 < a^2 < M/2$, we find a first approximation, x_1, by finding an integer, p, such that $2^{p-1} < a^2 \leq 2^p$. Since one of the integers, p and (p - 1), must be even, it follows that $2^{\frac{p-1}{2}} \leq x_1 = 2^{[p/2]} \leq 2^{p/2}$. Moreover, we know from the definition of p that $2^{\frac{p-1}{2}} < a \leq 2^{p/2}$; therefore, $|x_1 - a| \leq 2^{\frac{p-1}{2}} \cdot (2^{1/2} - 1) < 2^{\frac{p-3}{2}}$.

Proceeding as in the derivation of the "optimal" division procedure in paragraph 4.1.5, we obtain more information about our first approximation by examining $e_1 = a^2 - x_1^2$. Obviously, $a > x_1$, $a = x_1$, or $a < x_1$, according as $e_1 > 0$, $e_1 = 0$, or $e_1 < 0$; and, by considering the above bounds on a and x_1, it follows that $|e_1| \leq 2^{p-1}$. Hence, e_1 cannot overflow.

We note that the differential of e_1 is given by $de_1 = -2x_1\, dx_1$. Therefore, to reduce the error to zero, we should have $de_1 = -e_1$ or, equivalently, $dx_1 = e_1/2x_1$, which is what is done in the Newton-Raphson formula. However, in order to avoid the division indicated, we depart from the Newton-Raphson formula by approximating $e_1/2x_1$ by the ratio of the power of two high approximations of e_1 and $2x_1$. We find an integer, z_1, such that $2^{z_1 - 1} < |e_1| \leq 2^{z_1}$ and correct by an amount equal to $2^{z_1 - p'}$, where $p' = \left[(p + 1)/2\right] + 1$. Generalizing this technique to form an iterative procedure, we define $x_i = x_{i-1} \pm k_{i-1}$, taking the + or - sign according as $e_{i-1} > 0$ or < 0; $k_i = 2^{z_i - p'}$; $e_i = a^2 - x_i^2$; and if $e_i \neq 0$, z_i is an integer such that $2^{z_i - 1} < |e_i| \leq 2^{z_i}$.

If desired, we can represent the x_i's in the form $y_i \cdot 2^j$ used in the "optimal" division procedure. In this way, we generate without division a sequence of integers $y_1, y_2, y_3, \ldots$, which converges to $2^{-j} \cdot |a|$.

In the appendix it is shown that at least 1 binary bit of accuracy is attained per 2 iterations. In all examples calculated, the convergence is considerably faster.

Like the division method discussed in paragraph 4.1.5.4, this square-root procedure can be used in a variety of forms. The following is a step-by-step outline of how it can be used to approximate the square root of a given integer, $a^2 \geq 0$, in the form $y \cdot 2^i$, where i is a nonpositive number specified by the user.

Step 1 - Convert a^2 to TSMR and determine its sign. If $a^2 < 0$, set error indicator and terminate. If $a^2 = 0$, set $y = j = 0$ and terminate. Otherwise, use the table of powers of two to find an integer p such that $2^{p-1} < a^2 \leq 2^p$. If p is odd, go to Step 2; otherwise, go to Step 2a.

Step 2 - Set $j_{min} = \frac{p+1}{2} - n$ and compare i with j_{min}. If $i < j_{min}$, set error indicator and terminate. Otherwise, set $u = \frac{p - n - 1}{2}$, $j = \max(i, u)$, $q = \frac{p+1}{2}$, and $x = 2^{q - j - 1}$ Go to Step 3.

Step 2a - Set $j_{min} = \frac{p}{2} - n$ and compare i with j_{min}. If $i < j_{min}$, set error indicator and terminate. Otherwise, set $u = \left[\frac{p - n}{2}\right]$, $j = \max(i, u)$, $q = p/2$, and $x = 2^{q - j}$. Go to Step 3.

Step 3 - Calculate $e = 2^{-2j} \cdot a^2 - x^2$ and convert e to TSMR. If $e = 0$, go to Step 5; otherwise, change the sign of e, if necessary, to obtain $|e|$ and find an integer z such that $2^{z-1} < |e| \leq 2^z$. Calculate $w = z - q - 1 + 2j - i$. If $w \geq 0$, go to Step 4; otherwise, go to Step 5.

Step 4 - Calculate $u = \left[\frac{n - z + 1}{2}\right]$ and set $v = \min(u, j - i)$. Replace x by $2^v \cdot x \pm 2^{z - q - 1 + j + v}$, taking the + or - sign according as $e > 0$ or $e < 0$. Decrease j by v and go to Step 3.

Step 5 - Set $y = 2^{j-i} \cdot x$, $j = i$, and terminate. The error will be less than 2^{i-1}.

Example: Let us take $a^2 = 627,323$, $i = -11$, and $n = 22$. Then, in Step 2 we get $j_{min} = -12$; $i, v = 1$ and $q = 10$. Hence, our first approximation is $2048 \cdot 2^{-1}$. The results of the successive iterations are:

Iteration 1	Iteration 2	Iteration 3
$x = 2048$	$x = 3072$	$x = 6400$
$j = -1$	$j = -2$	$j = -3$
$e = -1,685,012$	$e = 599,984$	$e = -811,328$
$z = 21$	$z = 20$	$z = 20$
$w = 19$	$w = 16$	$w = 14$
$u = v = 1$	$u = v = 1$	$u = v = 1$

Iteration 4	Iteration 5	Iteration 6
$x = 12,672$	$x = 202,760$	$x = 405,524$
$j = -4$	$j = -8$	$j = -9$
$e = 15,104$	$e = 622,528$	$e = -754,064$
$z = 14$	$z = 20$	$z = 20$
$w = 6$	$w = 4$	$w = 2$
$u = v = 4$	$u = v = 1$	$u = v = 1$

Iteration 7
$x = 811,046$
$j = -10$
$e = 227,932$
$z = 18$
$w = -2$

Since $j > i$ in Iteration 7, Step 5 alters the answer to $1,622,092 \cdot 2^{-11}$.

To better illustrate the convergence in this example, we have constructed the following table in which the error is measured by $(x \cdot 2^j)^2 - a^2$.

Iteration	x	j	$x \cdot 2^j$	$(x \cdot 2^j)^2$	Error
1	2048	-1	1024	1,048,576	+421,253
2	3072	-2	768	589,824	- 37,499
3	6400	-3	800	640,000	+ 12,677
4	12,762	-4	797.625	636,205.64	+ 8,882.64
5	202,760	-8	792.030125	627,311.68	- 11.32

Iteration	x	j	$x \cdot 2^j 2$	$(x \cdot 2^j)^2$	Error
6	405,524	-9	792.0390625	627,325.88	+2.88
7	811,046	-10	792.0371093	627,322.78	-0.22
	1,622,092	-11	"	"	"

A preliminary calculation of the number of modular arithmetic operations required to complete the above procedure was compared with the number of operations required to achieve the same accuracy with the Newton-Raphson method. The results indicated that the above method required less than one-third the amount of calculation required by Newton-Raphson method.

A brief investigation was made into whether or not the above square-root procedure can be extended to permit extraction of "higher" roots, but the results were not encouraging enough to warrant further study.

4.1.7 Floating-Point Operations

An obvious extension of the basic idea behind the division procedure given in paragraph 4.1.5.4 - that is, representing the fraction, a/b, in the form $x \cdot 2^j$, where x is in "residue" form and j is an integer ≤ 0, carried in a special register - is "floating-point" arithmetic, in which operations such as addition, subtraction, multiplication, etc are performed in numbers in the form $x \cdot 2^j$ given above. All that is necessary is to remove the restriction that j be nonpositive and the result is a greatly increased machine range which gives modular arithmetic much more flexibility and power in performing complex calculations. However, this increased computing power comes at the cost of more complex - and hence, slower - addition, subtraction, and similar operations. This increased complexity can be kept to a minimum by requiring that the floating-point operands be expressed in a certain form, commonly called "normalized" form.

In the following, we shall say that any nonzero number given in the form $x \cdot 2^j$ is "normalized" if $2^{m-1} < |x| \leq 2^m$, where m is an integer defined by $2^m \leq \sqrt{M/2} < 2^{m+1}$. (A normalized "zero" will be represented by $0 \cdot 2^0$.) The choice of this criterion is somewhat arbitrary; the principal advantage is that it minimizes the amount of scaling necessary to prevent overflow in the "modular" parts of floating-point addition, subtraction, and multiplication. Another possible criterion for normalization might require that $2^{n-1} < |x| \leq 2^n$, where n is an integer such that $2^n \leq M/2 < 2^{n+1}$. This criterion gives a maximum number of significant digits in every operation but requires considerably more complicated addition and multiplication routines than the chosen criterion.

We shall now describe the detailed operation of each of a set of floating-point operations which assume that all operands (except where specifically stated otherwise) are normalized. These operations will make free use of a stored table of powers of two, the "nearest integer" variation of the division procedure, and the variation of the square root procedure given in paragraph 4.1.6 above. For the present, we assume that the exponent, j, must remain between $\pm h$, where h is some arbitrary bound, and that means are provided for detecting "overflow" ($j > h$) and "underflow" ($j < -h$). We also assume $h \geq n - m$.

4.1.7.1 Fixed to Floating-Point Conversion

This operation is essentially a "normalize" operation. Given a in modular form, we seek x and j such that x is normalized and $a = x \cdot 2^j$.

Step 1 - Convert a to TSMR and determine sign. If $a = 0$, set $x = j = 0$ and terminate; otherwise, set sign (x) = sign (a). Change signs of the TSMR coefficients of a, if necessary, so as to obtain the coefficients of $|x|$. Use the table of powers of two to find an integer p such that $2^{p-1} < |a| \leq 2^p$. Go to Step 2, 3, or 4 according as $p < m$, $p = m$, or $p > m$.

Step 2 - Let $|x| = |a| \cdot 2^{m-p}$, $j = p - m$, and terminate.

Step 3 - Let $|x| = |a|$, $j = 0$, and terminate.

Step 4 - Let $b = 2^{p-m}$ and perform "nearest integer" division of $|a|$ by b. Set $|x|$ equal to the result, $j = p - m$, and terminate. Note that since we assumed $h \geq n - m$, the exponent, j, cannot "overflow" nor "underflow" in the above operations.

4.1.7.2 Floating to Fixed-Point Conversion

Given $x \cdot 2^j$, we wish to find the residues of a such that $a \equiv x \cdot 2^j \pmod{M}$ if $j \geq 0$ and such that a is the "nearest integer" to $x \cdot 2^j$ if $j < 0$.

Step 1 - Determine the sign of j. If $j \geq 0$, set $a = x \cdot 2^j$, performing the multiplication in "modular" fashion, and terminate. Otherwise, go to Step 2.

Step 2 - Let $b = 2^{-j}$ and perform "nearest integer" division of x by b. Set a equal to the result and terminate.

Note that, if desired, an optional overflow detection could be performed during Step 1 to ascertain if $\left|x \cdot 2^j\right| > M/2$. The overflow detection method given in paragraph 4.1.2.2 would suffice.

4.1.7.3 Floating-Point Magnitude Comparison

Given $a = x \cdot 2^j$ and $b = y \cdot 2^k$ in normalized form, we wish to determine the larger of a and b.

Step 1 - Convert x and y to TSMR and determine the sign of each. If a and b are nonzero and sign (a) = sign (b), go to Step 2. Otherwise, if both a and b are nonzero, then $a > b$, or $a < b$ according as sign (a) = + or sign (a) = -. If either oı a or b is zero, say b, then $a > b$, $a = b$, or $a < b$ according as sign (a) = +, $a = 0$, or sign (a) = -. Terminate.

Step 2 - Compare j and k. If $j = k$, go to Step 3. Otherwise, $a > b$ if either $j > k$ and sign (a) = + or $j < k$ and sign (a) = -. Otherwise, $a < b$. Terminate.

Step 3 - Compare the TSMR coefficients of x and y in the usual manner to determine the greater of x and y. Then $a > b$, $a = b$, or $a < b$ according as $x > y$, $x = y$, or $x < y$. Terminate.

Note that sign detection is the same in both fixed and floating-point since sign (a) = sign (x) in the above.

4.1.7.4 Floating-Point Add and Subtract

Given $a = x \cdot 2^j$ and $b = y \cdot 2^k$, we wish to calculate $c = z \cdot 2^i$ such that $c = a \pm b$.

Step 1 - Compare j and k. If $j = k$, go to Step 4; otherwise, compare $r = |j - k|$ and m. If $r > m$, go to Step 2; otherwise go to Step 3.

Step 2 - If addition, set $c = a$ (i.e., $z = x$ and $i = j$) or $z = b$ according as $j > k$ or $j < k$. If subtraction, set $c = a$ or $c = -b$ according as $j > k$ or $j < k$. Terminate.

Step 3 - If $j < k$, do 3a; otherwise, do 3b. 3a - Set $w = y \cdot 2^r$, $t = j$ and $v = x \pm w$. Go to Step 5. 3b - Set $w = x \cdot 2^r$, $t = k$ and $v = w \pm y$. Go to Step 5.

Step 4 - Set $w = x \pm y$, and $t = j$, go to Step 5.

Step 5 - Convert w to TSMR and determine the sign of w. If $w = 0$, set $i = 0$ and terminate; otherwise, set sign (z) = sign (w) and find p such that $2^{p-1} < |w| \leq 2^p$. Go to Step 6, 7 or 8 according as $p < m$, $p = m$, or $p > m$.

Step 6 - Set $|z| = |w| \cdot 2^{m-p}$, $i = t + m - p$, and terminate, turning on underflow indicator if i underflows.

Step 7 - Set $|z| = |w|$, $i = t$, and terminate.

Step 8 - Set $i = t + p - m$, turning on overflow indicator and terminating if necessary. Set $u = 2^{p-m}$ and use "nearest integer" division to divide w by u. Let $|z|$ equal the result and terminate.

4.1.7.5 Floating-Point Multiply

Given $a = x \cdot 2^j$ and $b = y \cdot 2^k$, we wish to calculate $c = z \cdot 2^i$ such that $c = ab$.

Step 1 - Calculate $t = j + k$. If $|t| > |h|$, set overflow indicator and terminate. Otherwise, calculate $w = xy$ and convert w to TSMR. If $w = 0$, set $z = i = 0$ and terminate. Use the table of powers of two to find an integer p such than $2^{p-1} < |w| \leq 2^p$. Set $i = t + p - m$, turning on overflow indicator and terminating if t overflows. If no overflow occurs, go to Step 2.

Step 2 - Set $b = 2^{p-m}$ and use "nearest integer" division to divide w by b. Set z equal to the result and terminate.

4.1.7.6 Floating-Point Square-Root

Given $a = x \cdot 2^j, \geq 0$, we wish to find $b = y \cdot 2^k \geq 0$ such that $b^2 = a$. We assume for convenience that m is odd; if m is even, minor modifications must be made in Step 1.

Step 1 - Test j to see whether it is odd or even. If even, let $w = x$, $s = 0$, and $t = j$ and go to Step 2; if odd, let $w = 2x$, $s = 1$, $t = j - 1$, and go to Step 2.

Step 2 - Use the square-root procedure to find $\sqrt{w}$. Specify $i = \frac{s-m}{2}$. Set y equal to the "modular" portion of the result if it is nonzero and set $k = t/2 + i$; otherwise, set $y = k = 0$. Terminate.

If the square-root procedure referred to in Step 2 sets an error indicator, then $a^2 < 0$.

4.1.7.7 Floating-Point Divide

Given $a = x \cdot 2^j$ and $b = y \cdot 2^k$, we wish to find $c = z \cdot 2^i$ such that $c = a/b$.

Step 1 - Convert a and b to TSMR and determine the signs of both. If $b = 0$, set error indicator and terminate. If $a = 0$, set $z = i = 0$ and terminate. Otherwise, go to Step 2.

Step 2 - Set sign (z) = + if a and b have the same sign, and set sign (z) = - otherwise. Compare $|a|$ and $|b|$. If $|a| \leq |b|$, set $t = m$; otherwise, set $t = m - 1$ Set $s = \max(t, m - n - 1)$, $r = 2^s$, and go to Step 3.

Step 3 - Calculate $e = 2^{-s} |a| - |b| r$ and convert e to TSMR. If $e = 0$, go to Step 6; otherwise, find z such that $2^{z-1} < |e| \leq 2^z$. Calculate $w = z - m + 1 + s - t$. Go to Step 4, 5, or 6 according as $w > 0$, $w = 0$, or $w < 0$.

Step 4 - Calculate $u = z - n - 1$ and set $v = \max(s - t, v)$. Replace r by $2^v \cdot r \pm 2^{z-m+v}$, taking the + or - sign according as $e > 0$ or $e < 0$. Decrease s by v and go to Step 3.

Step 5 - Replace r by $r \cdot 2^{s-t}$ and set $s = t$. Calculate $e' = e \cdot 2^{s-t+1}$, convert e to TSMR, and obtain $|e'|$. Compare $|e'|$ and $|b|$. If $|e'| \geq |b|$, set $|z| = r \pm 1$ taking the + or - sign according as $e' > 0$ or $e' < 0$; if $|e'| < |b|$, set $|z| = r$. Go to Step 7.

Step 6 - Set $|z| = r \cdot 2^{s - t}$ and $s = t$. Go to Step 7.

Step 7 - Set $i = j - k + s$ and terminate.

Example: To illustrate the above operations, let us then solve $x^2 - 4x - 7 = 0$ for its greater root, $2 + \sqrt{11} = 5.3166$; that is, we will use "floating-point" arithmetic to evaluate the expression $\frac{-b + \sqrt{b^2 -4ac}}{2a}$. As in previous examples, we assume $n = 22$ and $M/2 = 4,849,845$. Hence, $m = 11$.

Operations 1, 2, and 3 - Convert 2a, 2c, and b to "floating-point."

$2a = 2 = 2048 \cdot 2^{-10}$

$2c = -14 = -1792 \cdot 2^{-7}$

$b \;= -4 = -2048 \cdot 2^{-9}$

Operation 4 - Multiply $2a \cdot 2c = (2048 \cdot 2^{-10}) \cdot (-1792 \cdot 2^{-7})$. Following the steps of paragraph 4.1.7.6 above, we have $t = -17$, $w = 3,670,016$, $p = 22$, $i = -6$, which gives $4ac = -1792 \cdot 2^{-6}$.

Operation 5 - Calculate $b^2 = (-2048 \cdot 2^{-9})^2$. Proceeding as in Operation 4, we have $t = -18$, $w = 4,194,304$, $p = 22$, $i = -7$. Hence, $b^2 = 2048 \cdot 2^{-7}$.

Operation 6 - Subtract 4ac from b^2; i.e., $b^2 -4ac = (2048 \cdot 2^{-7}) - (-1792 \cdot 2^{-6})$. Using paragraph 4.1.7.5 above gives $r = 1$, $w = -1792 \cdot 2^1 = -3584$, $v = 2048 + 3584 = 5632$, $t = -7$, $i = -5$. The result is $b^2 -4ac = 1408 \cdot 2^{-5}$.

Operation 7 - Find the positive square root of $b^2 -4ac = 1408 \cdot 2^{-5}$. From paragraph 4.1.7.8 above, we have $w = 2816$, $t = -6$, $i = -5$. The square root of w is $1698 \cdot 2^{-5}$; therefore, $\sqrt{b^2 -4ac} = 1698 \cdot 2^{-8}$.

Operation 8 - Add -b and $\sqrt{b^2 -4ac} = (2048 \cdot 2^{-9}) + (1698 \cdot 2^{-8})$. The procedure in paragraph 4.1.7.5 above gives $r = 1$, $w = 1698 \cdot 2^1 = 3396$, $t = -9$, $v = 2048 + 3396 = 5444$, $i = -7$. The result is $1361 \cdot 2^{-7}$.

Operation 9 - Divide $-b + \sqrt{b^2 -4ac} = (1361 \cdot 2^{-7})$ by $2a = 2048 \cdot 2^{-10}$. Paragraph 4.1.7.7 above gives $t = 11$, $s = -11$, $r = 2048$. The answer is $1361 \cdot 2^{-8}$. Hence, $x = 1361 \cdot 2^{-8} = 5 \cdot 3164$.

4.2 PROBLEM APPLICATIONS

In this paragraph we consider several significant problems which are often solved on computers. Our purpose is to determine any particular advantage or difficulties in solving these problems in modular arithmetic. In general, where the method usually used to solve a problem encounters difficulties in its adaptation to modular arithmetic, we have tried to find a new approach to the solution which is based on the specific properties of residue number systems.

4.2.1 The Laplace Equation

The first problem considered is that of solving the Laplace equation subject to prescribed boundary values by the method of nets (or finite differences). In this method, each interior point is unknown and has four neighboring values and each boundary point is known. Initial values are assigned to the interior points as the best guess which can be made. A new value is then calculated for each interior point which constitutes a second approximation to the solution. These values are reached by adding the former values of the four nearest neighbors together and dividing by 4. This process is continued to produce successively closer approximations to the solution.

Since it is necessary in modular arithmetic to work only with integers, a slight adaptation is needed in this method. The boundary values are assumed to be given as fractions and can be converted into integers by appropriate shifts, amounting to a change of variable. Once the boundary values are integers, the chosen initial values of the interior points can be integers. All values are then translated into modular numbers. From this point, the only difference in the modular procedure over the non-modular is that after adding together the four neighboring points to produce a new approximation, division by four is not performed. Instead, the boundary values are multiplied by 4 after each new approximation. After the modular values are translated to binary form, they are right shifted two places for each successive approximation calculated. A final shift takes place to compensate for the change of variable introduced initially.

A hand calculation of the boundary value problem on a 3 x 3 grid was carried out, but the conclusion was that results of any value in determining rates of convergence, etc., could only be carried out by computer simulation. However, because the use of modular arithmetic presents no peculiar difficulties or departures from those usually encountered in solving the Laplace equation on digital computers, no simulation was attempted.

4.2.2 Linear Differential Equations

As a second problem, we consider the solution of linear differential equations. In particular, we shall analyze a typical differential equation to see what sort of overflow and roundoff problems occur in its solution. In order to handle noninteger quantities, we shall replace the dependent variable with the ratio of two integer variables; therefore, two recursion relations must be used. The equation to be analyzed is:

$$\frac{dx}{dt} = tx + \frac{1}{2 - t}, \quad 0 \leq t \leq 1. \tag{25}$$

Let the step size be $1/q$, and solve iteratively using the first two terms of the Taylor series expansion for $x\,(n/q + 1/q)$; namely,

$$x_{n+1} = x_n + \frac{1}{q}\left(n\, x_n + \frac{1}{2 - n/q}\right),$$

where

$$x_n = x\,(nq).$$

Replace x_n by the ratio of y_n to z_n. The recursion formuli for these are

$$y_{n+1} = (q^2 + n)\,(2q - n)\,y_n + q^2 z_n, \tag{26}$$

$$z_{n+1} = q^2\,(2q - n)\,z_n, \quad 0 \leq n \leq q - 1.$$

If $x_o = 0$, $y_o = 0$. We may choose $z_o = 1/q^2$ and actually start the recursion with $y_1 = 1$ and $z_1 = 2q$. The magnitude of the final result is dismaying, even for small q, for

$$z_{n+1} > q^{3\,(n+1)}\, z_o = q^{3n+1}. \tag{27}$$

For $q = 10$, $z_{10} > 10^{28}$.

Clearly, to obtain useful results, we must have some sort of control of overflow. Monitoring the output and recognizing that discontinuities occur where overflow occurs might be sufficient but this approach has not yet been pursued in detail. Another approach is to scale down both y_n and z_n throughout the solution process so as to leave their ratio substantially unchanged. The frequency of scaling required will depend on the number range, M, and the step size, $1/q$. Since the solution of the differential equations is positive, the number range can be regarded as (0, M). To prevent overflow of y_{n+1} or z_{n+1}, restrict y_n and z_n as follows:

$$y_{n+1} = (q^2 + n)(2q - n)\, y_n + q^2 z_n < M \tag{28}$$

$$z_{n+1} = q^2 (2q - n)\, z_n < M. \tag{29}$$

From equation 28, $2(q + 1)\, y_n + z_n < M/q^2$ will ensure that $y_{n+1} < M$. From equation 29, $z_n < M/q^3$, a constant, will ensure that $z_{n+1} < M$. The inequality,

$$y_n < \frac{M(q - 1)}{2(q + 1)\, q^3},$$

a constant, and the constraint on z_n are therefore sufficient conditions for $y_{n+1} < M$. These two checks can be made at the end of each iteration, and the necessary scaling can be effected by using the technique of paragraph 4.1.3 of this report. Scaling by a factor m may introduce an error as large as

$$\max \left\{ \left| \frac{m}{z_n - m} \right|, \left| \frac{y_n}{z_n} \cdot \frac{m}{z_n - m} \right| \right\}$$

for least positive residues, or

$$\max \quad \frac{m}{2z_n - m}, \ \frac{y_n}{z_n} \cdot \frac{m}{2z_n - m}$$

for least absolute value residues. From these error estimates, it appears that least absolute value residues are preferable.

It can be shown that an error at step p of the recursion propagates to result at step n in an error

$$e_n = e_p \prod_{j=p}^{n-1} \left(1 + \frac{j}{q^2}\right).$$

Hence, the truncation error and roundoff error for a particular step are given by

$$E_t(n) = \frac{q^2 + n^2}{2n^4} + \frac{q^2 + 2qn - n^2}{2q^2 (2q - n)^2} \tag{30}$$

and

$$E_r(n) = \frac{q^3}{m - q^3},$$

where the truncation error is taken as the second derivative term of the Taylor series.

The total accumulated errors due respectively to truncation and round-off at each step are

$$E_t = \sum_{n=0}^{q-1} \left\{ \frac{q^2 + n^2}{2n^4} + \frac{q^2 + 2qn - n^2}{2q^2\,(2q - n)^2} \right\} \prod_{j=n}^{q-1} \left(1 + \frac{j}{n^2}\right) \tag{31}$$

$$\left|E_r\right| \leq \frac{q^3}{m - q^3} \sum_{j=n}^{q-1} \prod_{j=n}^{q-1} \left(1 + \frac{j}{n^2}\right)$$

and are bounded by

$$E_t < e^{\frac{q-1}{2q}} \left[\frac{(q-1)(2q-1)}{12\,q^3} + \frac{1}{q}\left(\ln 2 - \frac{1}{4q}\right) + \frac{1}{2}\left(\frac{1}{4q} + \frac{q}{2\,(q+1)\,(2q+1)}\right) \right]$$

$$\left|E_r\right| < \frac{e^{\frac{q-1}{2q}}\, q^3}{m - q^3} \left[2 + \sqrt{\pi}\; q \operatorname{erf}\left\{ \frac{\sqrt{2}\,(q-2)}{q} \right\} e^{\frac{(q-2)^3}{3q^4}} \right]^{1/2} \tag{32}$$

The above bound for the effect of roundoff error is based on the assumption that this error is always maximum and of the same sign. Assuming a uniform distribution of roundoff error, it is easy to show that the rms error is $1/\sqrt{3}$ of that given above.

The following table shows the values of the error bounds for various values of q and m. Notice that the largest contribution to the overall root-square error comes from the truncation error. A higher order integration method might bring the two error contributions closer together and reduce the total error.

Steps q	Range m	Truncation Bound T	Roundoff Bound R	$\sqrt{T^2 + R^2}$
10	10^4	0.526	0.306	0.691
10^2	10^9	$1.813(10)^{-2}$	$9.24(10)^{-3}$	$2.03(10)^{-2}$
10^2	10^{10}	$1.813(10)^{-2}$	$9.24(10)^{-4}$	$1.81(10)^{-2}$
10^3	10^{13}	$1.830(10)^{-3}$	$2.60(10)^{-3}$	$3.18(10)^{-3}$
10^3	10^{14}	$1.830(10)^{-3}$	$2.60(10)^{-4}$	$1.85(10)^{-3}$
10^4	10^{17}	$1.830(10)^{-4}$	$8.22(10)^{-4}$	$8.42(10)^{-4}$
10^4	10^{18}	$1.830(10)^{-4}$	$8.22(10)^{-5}$	$2.01(10)^{-4}$
10^4	10^{19}	$1.830(10)^{-4}$	$8.22(10)^{-6}$	$1.84(10)^{-4}$

4.2.3 Complex Arithmetic

An interesting feature of modular arithmetic is that it lends itself well to working with complex (Gaussian) integers, $a + bi$, where a and b are ordinary integers and $i = \sqrt{-1}$. This fact may be useful in solving eigenvalue problems, where we often must compute with matrices of complex numbers.

To reduce a complex number (mod p), we reduce both a and b (mod p). Thus, for example, $19 - 11\,i \equiv 4 + 4i \equiv -1 - i \pmod{5}$. So long as we restrict ourselves to the operations of addition and multiplication it is clear that the corresponding equations (mod p) are also valid, and give the real and imaginary parts (mod p) of the result. However, it is also true that division can be carried out (in the same sense as with real integers) when the quotient is an integer (or in a Gaussian elimination), provided we work with moduli p which are primes and satisfy $p \equiv -1 \pmod{4}$; i.e., such primes as 3, 7, 11, 19, 23, ... The reason for this restriction is that the Gaussian integers (mod p) are a field (system in which division is possible) if and only if $p \equiv -1 \pmod{4}$.

The proof of this is based on the well-known theorem from the theory of numbers that -1 is a quadratic residue of p if $p \equiv 1 \pmod{4}$, and a quadratic nonresidue if $p \equiv -1 \pmod{4}$. Suppose we consider the number $a + bi$ where a and b are taken (mod p), and not both zero; i.e., $a + bi \not\equiv 0 \pmod{p}$. The condition that an inverse element, call it $x + iy$, exist, is that $(a + bi)(x + yi) \equiv 1 \pmod{p}$ or,

$$a\,x - b\,y \equiv 1 \tag{33}$$

and

$$b\,x + a\,y \equiv 0 \pmod{p}.$$

These congruences have a solution if and only if their determinant $a^2 + b^2 \not\equiv 0 \pmod{p}$. If now $p \equiv -1 \pmod{4}$ then $a^2 + b^2 \not\equiv 0$ mod p, for otherwise $-a^2 = b^2$ is a quadratic residue which contradicts the above theorem. On the other hand, if $p \equiv 1 \pmod{4}$, $a^2 + b^2 \equiv 0 \pmod{p}$ has a solution with $a \not\equiv 0$ (in fact, it is easily shown that p itself is a sum of two squares), and $a + bi$ has no inverse (mod p).

Example:

$$\frac{2 + i}{3 - 2i} \equiv \frac{(2 + i)(3 + 2i)}{13} \equiv \frac{4}{13} \equiv 3 \pmod{7}.$$

This enables us to extend all of the linear algebra algorithms of modular arithmetic to the complex domain. To carry out the complex operations (mod p) either a separate subroutine can be included in the (mod p) unit, or else the whole closed system of p^2 elements (the so-called Galois Field GF $[p^2]$ can be mechanized as a basic unit, which does ordinary (i.e. real) mod p arithmetic as a subroutine.

Let us give a simple numerical example.

Calculate the determinant

$$\begin{vmatrix} 2+i & 5 & -i \\ 1+i & 1-i & 2 \\ 1 & -3i & -10+i \end{vmatrix}.$$

(mod 7) by Gaussian elimination.

Note $(2+i)^{-1} \equiv -1 -3i$. Multiplying the first row successively by (1+i) (1+3i) and by (1+3i) and adding successively to the second and third rows gives

$$\begin{vmatrix} 2+i & 5 & -1 \\ 0 & 4-i & 4+2i \\ 0 & -2-2i & 0 \end{vmatrix}.$$

Finally, $(4-i)^{-1} \equiv -1-2i$; hence, to complete the triangularization multiply the second row by $(1+2i)(-2-2i) \equiv 2+i$ and add to the third row, obtaining

$$\begin{vmatrix} 2+i & 5 & -i \\ 0 & 4-i & 4+2i \\ 0 & 0 & -1+i \end{vmatrix}.$$

Hence, the value of tne determinant is $(2+i)(4-i)(-1+i) \equiv 3 \pmod 7$. In other words, the determinant has the form, $a + bi$, where $a \equiv 3$ and $b \equiv 0 \pmod 7$.

4.2.4 Polynomial Evaluation

It has been pointed out that redundancy inherent in various numerical calculations takes the form of periodicity in modular arithmetic, thus reducing the amount of computation required. In the following, it is shown how the evaluation of a polynomial for all integers, x, such that $0 \leq x < M$ = the product of the moduli, is reduced to the evaluation of the same polynomial for only m_n values, where m_n is the largest modulus in the system.

The most general polynomial, P(x), which takes on integer values for all integer values of x may be written

$$P(x) = \sum_{i=0}^{N} a_i \binom{x}{i}, \tag{34}$$

where

$$\binom{x}{i} = \frac{x(x-1)\ldots(x-i+1)}{i!},$$

a binomial coefficient, and the a_i are integers. A formula for the period of such a polynomial modulo m appeared in a recent issue of the American Mathematical Monthly (reference 4).

In practice, however, one is more likely to be concerned with a polynomial which has integer coefficients. If, furthermore, all moduli are primes, p, the problem is considerably simplified. Then p is a period of the polynomial modulo p, so that either the polynomial is constant modulo p or has fundamental period p.

Again, in practice, it doesn't appear worthwhile to determine whether the polynomial is constant or not for any particular modulus since the evaluation may proceed simultaneously in all moduli. The number of evaluation cycles required is then exactly equal to the largest modulus. In modular notation, the value of the polynomial for any integer x in $0 \leq x < m_1 m_2 \ldots m_n$, where $m_1, \ldots, m_n$ are the (prime) moduli is $\left[P(x_1), P(x_2), \ldots, P(x_n)\right]$ where x in modular notation is $(x_1, x_2, \ldots, x_n)$.

In the evaluation of the polynomial, Fermat's Little Theorem in the form

$x^p = x$ modulo p

may be used to reduce the polynomial modulo p to one of degree (p-1) or lower. In this case, the polynomial is constant modulo p if and only if it is of degree 0.

4.2.5 Linear Algebra

4.2.5.1 The Adjoint of a Quasi-Singular Matrix

It is known (see reference 5) that, in the case of a singular matrix, A, the process of Gaussian elimination will lead to a row of zeros. The rows of the adjoint of A are proportional to the corresponding row $x = (x_1, x_2, \ldots, x_n)$ of the bookkeeping matrix.

Clearly, the transpose of A is also singular, so that Gaussian elimination performed upon it will also lead to a row of zeros. By transposing the corresponding row $y = (y_1, y_2, \ldots, y_n)$ of the bookkeeping matrix, one obtains a column vector. The columns of the adjoint of A are proportional to this column vector. (Alternatively, one may perform elementary column operations on A and on a bookkeeping matrix to obtain a column of zeros and the corresponding column of the bookkeeping matrix.)

The adjoint of A is, except for a scalar factor, the product of the column and row vectors; that is, the ijth element of the adjoint of A is $cy_i x_j$. The constant c can be evaluated most directly as follows. Let y_s and x_r be the first nonzero elements of y and x, respectively. Then, $cy_s x_r = (-1)^{r+s} a_{rs}'$, where a_{rs}' is the value of the minor determinant of the rs th element of A. The above equation is actually a congruence modulo p which can be solved for c since $y_s x_r \not\equiv 0$ (mod p). The evaluation of the determinant a_{rs}' (mod p) is accomplished in the usual manner for nonquasi-singular matrices. If $a_{rs}' \equiv 0$ (mod p), then $c \equiv 0$ and the adjoint of A is identically zero modulo p.

4.2.5.2 Gauss Elimination for Singular Matrices

In paragraph 4.2.5.1, a modular method for finding the adjoint modulo p of a matrix whose determinant is congruent to 0 mod p (a quasi-singular matrix) was described. This method has the disadvantage of requiring significantly different processing of the matrix mod p compared to the processing relative to the other moduli. Furthermore, in the related but simpler problem of solving a set of linear equations, the more time consuming operation of finding the adjoint of the coefficient matrix was required as a preliminary, rather than direct solution of the system of equations. A new method of finding the adjoint of a singular matrix has been discovered; this is simply the addition of another operation, applicable to singular (or quasi-singular) matrices only, to the operations used in Gauss elimination. With this operation, Gauss elimination may also be used in solving all systems of linear equations. This method will now be described

To avoid confusion, one standard variant of the Gauss elimination process will be described and will hereafter be considered as the Gauss process. The extension works equally well with other variants; obvious minor modifications to the proof of its validity are required.

To invert the n x n matrix A the process is as follows: The n x n identity, I, is adjoined to A to form an n x 2n matrix (A, I). The nonsingular operations of row interchanges and elementary row operations (multiplication of a row by a nonzero constant and addition of a multiple of a row to another row) are performed to bring A first to upper triangular form (the forward course) and then to complete the transformation of A to the identity, I, (the return course). At this point, I in the augmented matrix has been transformed to A^{-1} and the determinant of A is the reciprocal of the product of the factors used as row multipliers times $(-1)^r$, where r is the number of row interchanges performed. The adjoint of A is given by $A^* = |A| A^{-1}$.

The forward course may always be completed; when A is singular, one or more rows of the transform of A will be 0. The return course, which starts with the nth row of A, and uses only elementary row operations (no row interchanges) can proceed only to the point where the addition of an (infinite) multiple of a zero row to other rows is required.

For definiteness, it is assumed that by this point, all diagonal elements of the transform of A are either 0 or 1. If not, the rows of the augmented matrix are each multiplied by the reciprocal of the corresponding nonzero diagonal element. The product, b, of the reciprocals of all the row multipliers and $(-1)^r$ is required. At this point then, the matrix (A, I) has been transformed to (T, S) where

$$\det S = 1/b \neq 0 \tag{35}$$

and T has at least one row of zeros. Since the same operations which transformed A to T transformed I to S,

$$SA = T. \tag{36}$$

The same process is followed to solve a set of linear equations except that the column y is adjoined to A to form the n x (n + 1) matrix (A, y). This process breaks down at the same point.

4.2.5.3 Extended Gauss Elimination

The extension is simply the addition of the following operation, E, to those previously used. When a row of zeros, say the i^{th}, is encountered in the transform of A, the diagonal element of that row is changed to 1, and in the augmented portion of the matrix all other rows are changed to 0, the i^{th} row being unchanged.

This operation, E, treats the portions T and S or T and Sy of the transformed augmented matrix dissimilarly. Let D_i be the n x n matrix with a 1 in the (i, i) position and 0 elsewhere. Then E is simply:

$$T \rightarrow T + D_i$$

$$S \rightarrow D_i\, S$$

or

$$(Sy) \rightarrow D_i\, (Sy)$$

Thereafter the ordinary Gauss process proceeds. Since all diagonal elements used in the Gauss process are 1, only the operation of addition of a multiple of one row to another is required. This operation has determinant 1.

If another 0 diagonal element, say the j^{th}, is encountered, the operation E is repeated. After a finite number of such operations (at most n) the augmented matrix of the inversion problem is in the form (I, W). Then,

$$b\,W = A^*.$$

For the problem of the solution of the set of linear equations, W is not displayed explicity; the augmented portion of the matrix is (Wy) and b (Wy) is the required solution.

Before demonstrating the validity of this method, some illustrative examples will be presented. Consider the problems of finding the adjoint of A and of solving $Ax \equiv y$ modulo 7. (Recall that we will actually find the integer vector (detA)x.)

$$A = \begin{pmatrix} 1 & 2 & 6 \\ 2 & 1 & 3 \\ 3 & 4 & 5 \end{pmatrix} \qquad \begin{aligned} 1 \cdot x_1 + 2x_2 + 6x_3 &= 1 \\ 2 \cdot x_1 + 1 \cdot x_2 + 3x_3 &= 2 \\ 3 \cdot x_1 + 4 \cdot x_2 + 5x_3 &= 2 \end{aligned}$$

$$\det A = 21 \equiv 0 \bmod 7.$$

$$(A, I) = \begin{pmatrix} 1 & 2 & 6 & 1 & 0 & 0 \\ 2 & 1 & 3 & 0 & 1 & 0 \\ 3 & 4 & 5 & 0 & 0 & 1 \end{pmatrix} \qquad (A, y) = \begin{pmatrix} 1 & 2 & 6 & 1 \\ 2 & 1 & 3 & 2 \\ 3 & 4 & 5 & 2 \end{pmatrix}$$

Subtracting mod 7 twice (3 times) the first row from the second (third) row, one obtains:

$$\begin{pmatrix} 1 & 2 & 6 & 1 & 0 & 0 \\ 0 & 4 & 5 & 5 & 1 & 0 \\ 0 & 5 & 1 & 4 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 6 & 1 \\ 0 & 4 & 5 & 0 \\ 0 & 5 & 1 & 6 \end{pmatrix}$$

To complete the forward course, the second row is multiplied by 2, whose reciprocal is 4 and five times the resulting row subtracted from the third.

$$(T, S) = \begin{pmatrix} 1 & 2 & 6 & 1 & 0 & 0 \\ 0 & 1 & 3 & 3 & 2 & 0 \\ 0 & 0 & 0 & 3 & 4 & 1 \end{pmatrix}_{b = 4} \qquad (T, Sy) = \begin{pmatrix} 1 & 2 & 6 & 1 \\ 0 & 1 & 3 & 2 \\ 0 & 0 & 0 & 6 \end{pmatrix}$$

Now the operation E is applied: the 0 in the third row, third column is changed to a 1 and the elements of the first two rows of the fourth and fifth and sixth columns if any, are changed to 0.

$$\begin{pmatrix} 1 & 2 & 6 & 0 & 0 & 0 \\ 0 & 1 & 3 & 0 & 0 & 0 \\ 0 & 0 & 1 & 3 & 4 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 6 & 0 \\ 0 & 1 & 3 & 0 \\ 0 & 0 & 1 & 6 \end{pmatrix}$$

The return course is now completed in the usual way; three times the last row is subtracted from the second row, six times the last row subtracted from the first row and twice the new second row from the new first row.

$$(I, W) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 5 & 2 & 4 \\ 0 & 0 & 1 & 3 & 4 & 1 \end{pmatrix} \qquad (I, Wy) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 6 \end{pmatrix}$$

Multiplying by b = 4

$$A^* = 4W = \begin{pmatrix} 0 & 0 & 0 \\ 6 & 1 & 2 \\ 5 & 2 & 4 \end{pmatrix} \qquad 4(Wy) = \begin{pmatrix} 0 \\ 5 \\ 3 \end{pmatrix}$$

checking, the nonmodular adjoint of A is

$$\begin{pmatrix} -7 & 14 & 0 \\ -1 & -13 & 9 \\ 5 & 2 & -3 \end{pmatrix}$$

which is congruent mod 7 to A^* above, and the solution to the set of equations is

$$21\, x_1 = 21 \qquad 21\, x_2 = -9 \qquad 21\, x_3 = 3$$

which are congruent mod 7 to 0, 5, and 3 as determined above.

It remains to prove that inclusion of the operation E in the above procedure leads to A^*.

From equation 36 it follows that

$$A^* S^* = T^*.$$

Multiplying on the right by S and using equation 35 yields

$$A^* = b\, T^* S.$$

Since the ith row of T is 0, the cofactor of each element of T not in the ith row is 0 so that T^* differs from 0 only in the ith column. But then $T^* S$ depends only on the ith row of S so that S may be replaced by $D_i S$ which leaves the ith row of S unchanged; thus,

$$A^* = b\, T^* (D_i S).$$

(This relation verifies the known result that the rows of A^* are proportional to the ith row of S.)

Since $D_i S$ differs from 0 only in the ith row, the product $T^* (D_i S)$ depends only on the ith column of T^*; this column is independent of the ith row of T so that T may be replaced by $T + D_i$. Hence

$$A^* = b\, (T + D_i)^* (D_i S) \tag{37}$$

If $T + D_i$ is nonsingular, Gauss elimination on the augmented matrix $(T + D_i, D_i S)$ can be completed leading to a matrix (I, W). Clearly the transformation of $T + D_i$ to I is accomplished by $(T + D_i)^{-1}$; since $T + D_i$ is a triangular matrix with diagonal elements 1, its determinant is 1 so that

$$(T + D_i)^{-1} = (T + D_i)^{*}$$

and

$$W = (T + D_i)^{*} (D_i S).$$

Substituting in equation 37,

$$A^{*} = b\,W$$

as required.

A proof for the case $T + D_i$ singular may be had for example by induction on the number of 0 diagonal elements of T. It may be noted that if 2 or more rows of the transform of A are simultaneously 0, say the ith and jth, then S may be replaced by

$$D_i\, D_j\, S = 0.$$

Since any linear transformation of 0 yields 0, the known result,

$$A^{*} = 0$$

when A is multiply degenerate, is obtained.

4.2.5.4 Computation of the Characteristic Polynomial

As a general rule, all linear algebra computations which are "rational"; i.e., do not involve solution of higher degree equations etc, extend without difficulty to MA. Next to inversion of a matrix, computation of the characteristic polynomial $\Delta(X) = \det(A - XI)$ ranks high in importance. This succeeds very elegantly by transformation to Frobenius normal form which we explain briefly (reference 6).

Let us denote by $\lambda\colon i \overset{R}{\rightarrow} j$ the row operation in which λ times the ith row is added to the jth row. The operation $\lambda\colon i \overset{R}{\rightarrow} j$ is realized as

left multiplication by the matrix $\begin{bmatrix} 1 \ddots & \vdots & \\ \cdots\cdots & \lambda \ddots & \cdots\cdots \\ & \vdots & 1 \end{bmatrix}$

where λ is in the (j, i) position. Similarly we let $\lambda\colon i \overset{C}{\rightarrow} j$ denote the addition of λ times the ith column to the jth column. This operation is realized as right multiplication by the matrix which is a unit matrix, except for λ in the (i, j) position. (It is assumed throughout that $i \neq j$.) Now, the operation inverse to $\lambda: i \xrightarrow{R} j$ is clearly $-\lambda: i \xrightarrow{R} j$. Representing this as a matrix, and multiplying by this inverse matrix on the right thus represents the operation $-\lambda: j \xrightarrow{C} i$. We thus arrive at the following important conclusion: The combined effect of adding λ times the i^{th} row to the j^{th} row, followed by addition of $-\lambda$ times the j^{th} column to the i^{th} column is a similarity transformation. Hence clearly we can perform any consecutive number of row operations on a matrix, and then "balance" these with the appropriate column operations (care must be taken with the order) so that the characteristic polynomial remains unchanged. A second important similarity transformation is obvious: multiply the i^{th} row by λ, then the i^{th} column by λ^{-1}. And a third, even simpler one, is: interchange rows i and j, then interchange columns i and j. The transformation to Frobenius normal form is based on bringing the given matrix, by the three basic similarity operations just described, to the form

$$\begin{bmatrix} c_1 & c_2 & \cdots & c_{n-1} & c_n \\ 1 & 0 & & 0 & 0 \\ 0 & 1 & & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

of which the characteristic polynomial is readily seen to be $\Delta(X) = (-1)^n (X^n - c_1 X^{n-1} - \ldots - c_n)$. (The factor $(-1)^n$ may be ignored; thus the coefficients of the characteristic polynomial are just the numbers $-c_i$.) The reduction to the form above by an algorithm now to be described can always be carried out unless we reach a form with a certain pattern of zeroes in which case calculation of the characteristic polynomial is thrown back upon the same problem for matrices of lower order. Let us illustrate the method numerically. Let

$$A = \begin{bmatrix} 3 & -1 & -4 & 2 \\ 2 & 3 & -2 & -4 \\ 2 & -1 & -3 & 2 \\ 1 & 2 & -1 & -3 \end{bmatrix}$$

It is required to compute $\Delta(X)$. Clearly for any prime, p, the coefficients of $\Delta(X)$ are $\equiv$ (mod p) to the coefficients of the polynomial obtained when the elements of A are replaced by residues (mod p). Let us choose $p = 5$. Then the transformations are as follows:

$$A \equiv \begin{bmatrix} 3 & 4 & 1 & 2 \\ 2 & 3 & 3 & 1 \\ 2 & 4 & 2 & 2 \\ 1 & 2 & 4 & 2 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 4 & 4 & 2 \\ 2 & 3 & 2 & 1 \\ 2 & 4 & 3 & 2 \\ 1 & 2 & 1 & 2 \end{bmatrix} \rightrightarrows \begin{bmatrix} 3 & 4 & 4 & 2 \\ 2 & 3 & 2 & 1 \\ 3 & 1 & 2 & 3 \\ 1 & 2 & 1 & 2 \end{bmatrix}$$

Here a single arrow denotes a single (row or column) transformation, a double arrow denotes that the matching (column or row) transformation has been performed. Thus any matrix following a double arrow is similar to A. We will employ the same convention for a series of row (or column) transformations performed simultaneously. Notice that we have a 1 in the (4, 3) position. This is the first step and now we are in a position to transform all other elements in row 4 to 0. Performing successively $-1 : 3 \xrightarrow{C} 1$, $-2 : 3 \xrightarrow{C} 2$, $-2 : 3 \xrightarrow{C} 4$ gives

$$\begin{bmatrix} 4 & 1 & 4 & 4 \\ 0 & 4 & 2 & 2 \\ 1 & 2 & 2 & 4 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

and now performing successively $1 : 1 \xrightarrow{R} 3$, $2 : 2 \xrightarrow{R} 3$, $2 : 4 \xrightarrow{R} 3$ gives

$$\begin{bmatrix} 4 & 1 & 4 & 4 \\ 0 & 4 & 2 & 2 \\ 0 & 1 & 2 & 2 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

We now bring a 1 into the (3, 2) position (it is already there in the present case) and use it to obtain zeroes in the remaining positions of row 3. These operations require no comment:

$$\begin{bmatrix} 4 & 1 & 2 & 2 \\ 0 & 1 & 1 & 4 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

at this point we cannot proceed owing to the zero in position (2, 1). Moreover, this zero is "irreducible," there is no nonzero element standing in its row to the left of it (otherwise we could bring that nonzero element in its place by successive row interchange, and column interchange without disturbing rows 3 and 4). This is the case when the algorithm stops, and we write

$$\Delta(X) = (4 - X) \det (A_1 - XI)$$

where

$$A_1 = \begin{bmatrix} 1 & 1 & 4 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

is in Frobenius normal form, and we have the result

$$\Delta(X) \equiv (X - 4)(X^3 - X^2 - X - 4)$$

$$\Delta(X) \equiv (X + 1)(X^3 - X^2 - X + 1) \equiv (X + 1)^2 (X - 1)^2 \pmod 5$$

In point of fact $\Delta(X) = (X + 1)^2 (X - 1)^2$ as direct calculation shows. The reader will find it instructive to work the same example for other prime moduli.

To determine the number of primes required, we reauire the following information: Let b_i denote positive numbers, and

$$(X + b_1) \ldots (X + b_n) = X^n + s_1 X^{n-1} + s_2 X^{n-2} + \ldots s_n \tag{38}$$

$$s_k = \binom{n}{k} t_k \tag{39}$$

Thus, s_k is the sum of the products of distinct b_i taken k at a time, and t_k is the average of these products. According to an old theorem of Maclaurin,

$$t_1 \geq t_2^{\frac{1}{2}} \geq \ldots t_n^{\frac{1}{n}} . \tag{40}$$

(For a proof see Hardy et al, Inequalities, Cambridge, 1934, p. 52.) A second fact which we require from the theory of matrices is the inequality

$$\sum_1^n |\lambda i|^2 \leq \sum_{i=1}^n \sum_{j=1}^n |a_{ij}|^2 \tag{41}$$

where λ_i represent the eigenvalues of the matrix $\|a_{ij}\|$. Now, in the characteristic polynomial $\Delta(X)$, the coefficient c_k of X^{n-k} is the sum of products of the λ_i taken k at a time. Now, in equation 38, let $b_i = |\lambda_i|$. Then $|c_k| \leq s_k$. Hence, since $t_k \leq t_1^k$ we have

$$s_k \leq \binom{n}{k} \frac{s_1^k}{n^k} \tag{42}$$

Hence

$$\left|c_k\right| \leq n^{-k} \binom{n}{k} \left(\Sigma \; \left|\lambda_i\right| \right)^k$$

$$\leq n^{-k} \binom{n}{k} n^{\frac{k}{2}} \left(\Sigma \; \left|\lambda_i\right|^2\right)^{\frac{k}{2}}$$

by Schwarz inequality. Hence finally

$$\left|c_k\right| \leq n^{-\frac{k}{2}} \binom{n}{k} \left(\sum_{i=1}^{n} \; \sum_{j=1}^{n} \left|a_{ij}\right|^2\right)^{\frac{k}{2}} \tag{43}$$

by equation 41. This gives us bounds on the coefficients of the characteristic polynomial. A simpler but more crude bound is obtained by replacing every a_{ij} by $a = \max \left|a_{ij}\right|$. This gives

$$\left|c_k\right| \leq n^{\frac{k}{2}} \binom{n}{k} a^k \tag{44}$$

Notice that when $k = n$, $c_k = c_n = \det A$, and equation 44 gives the Hadamard determinant inequality.

For example, suppose $a = 10$, $m = 5$. Then the upper bounds for $\left|c_1\right|$, $\left|c_2\right|$, $\left|c_3\right|$, $\left|c_4\right|$, $\left|c_5\right|$ given by equation 44 are respectively about 112; 5000; 112,000; 1,250,000; and 5,590,000. Since the upper bound given by equation 44 in most cases attains its largest value for $k = n$ (for instance, this is the case whenever $a^2 \geq n$) it is generally sufficient in computing the characteristic polynomial to use a set of primes sufficient for the computation of the determinant.

4.3 SUMMARY AND RECOMMENDATIONS FOR FUTURE WORK

In the above paragraphs of this section, we have shown how two-sided mixed-radix notation can be used to determine the sign of a number or to compare the magnitude of two numbers given in residue form, and we have given a method for converting an integer from residue notation to the two-sided mixed radix form. We have derived overflow detection methods for addition and multiplication, which in the superior version of the latter algorithm makes use of a stored table of powers of two. This table is also used to implement a division procedure which seems to be the best devised for modular arithmetic.

This division procedure, while slow compared to addition, subtraction, and multiplication on a modular arithmetic computer, is comparable in speed to the division operation on small, conventional computers. Hence, any conventional fixed-point program can be adapted, with no significant modifications, for modular arithmetic execution. Addition, subtraction and multiplication will be considerably faster on the MA computer, division will be about the same in speed, while sign determination is considerablv slower.

The division procedure can be extended to permit an essentially complete repertoire of floating-point operations. An extension of the idea of the division procedure yields a square root extraction algorithm.

We have also shown how modular arithmetic can be used to solve various commonly occurring problems such as Laplace's equation and linear differential equations. In the area of linear algebra, we have devised a new method for finding the adjoint of a matrix whose determinant is zero with respect to a modulus, a problem peculiar to modular arithmetic, and have applied this method to the solution of a set of linear equations. Finally, we have shown how the special properties of the modular arithmetic number system can be used to advantage in evaluating polynomials at many points and performing complex arithmetic. From these applications as well as the methods for extracting square roots and finding the adjoint of a matrix, it is apparent that it is quite often advantageous to utilize the special properties of the modular arithmetic number system in solving a problem, rather than merely to adapt the "standard methods" used on binary digital computers to the residue number system.

In the course of the investigations described above, we have found new problem areas which we feel are worthy of investigation in future numerical analysis work on modular arithmetic. Some of these problem areas are described below.

4.3.1 Scaling by Continued Fractions

It is well known (see reference 7) that a rational fraction, a/b, can be represented as a (finite) continued fraction and that this representation can be obtained through a repeated application of the Euclidean algorithm. This requires one "nearest-integer" division, one multiplication, and one subtraction per iteration, as is illustrated in the example below. The numerators and denominators of the successive convergents can be calculated through the use of integer recursion formulas. An investigation should be conducted as to the feasibility of scaling-down a fraction with a large numerator and/or denominator - such as occur in the solution of the linear differential equation in paragraph 4.2.2 - by approximating that fraction by one of the convergents in its continued fraction representation. The advantage of such a technique lies in the excellence of the approximation of the fraction by one of its convergents, considering the small size of the numerator and denominator of the approximation. The principal disadvantage is that the repeated application of the Euclidean algorithm requires time-consuming "nearest integer" divisions.

For example, we can expand 105/38 as a continued fraction as follows:

$$105 = 2(38) + 29$$
$$38 = 1(29) + 9$$
$$29 = 3(9) + 2$$
$$9 = 4(2) + 1$$
$$2 = 2(1).$$

Typically, the second line is obtained by $[38/29] = 1$; $38 - 1(29) = 9$.

If we label the successive "quotients" $q_1, q_2, \ldots,$ we have the result:

$s =$	0	1	2	3	4	5
$q_s =$		2	1	3	4	2
$P_s =$	1	2	3	11	47	105
$Q_s =$	0	1	1	4	17	38

where, for

$s \geq 1$,

P_s and Q_s are generated by the recursion formulas

$$P_s = q_s P_{s-1} + P_{s-2},$$

and

$$Q_s = q_s Q_{s-1} + Q_{s-2}.$$

Taking the q_s's in succession, we have the continued fraction representation of 105/38:

$$2 + \cfrac{1 \quad 1}{1 + \cfrac{1 \quad 1}{3 + \cfrac{1\,1}{4 + \cfrac{1}{2}}}}$$

Furthermore, we can approximate 105/38 by stopping at any value of $s < 5$ and using any of the "convergents" P_s/Q_s; it can be shown that the error in such an approximation is

$$\leq 1/Q_s Q_{s-1}.$$

Hence, we could use for example,

$$P_3/Q_3 = 11/4$$

as an approximation of 105/38.

4.3.2 Cyclic Orientation

Given the distinct integers a, b, and c, we say that the cycle (a, b, c) is positive if $a < b < c$ and is negative if $b < a < c$ or $a < c < b$. If a, b, and c are now represented in residue form by a_i's, b_i's and c_i's, respectively, we can perform certain operations on the rows and columns of the matrix

$$\begin{matrix} a_1 & a_2 & \cdots & a_n \\ b_1 & b_2 & \cdots & b_n \\ c_1 & c_2 & \cdots & c_n \end{matrix}$$

which preserve the orientation of the cycle (a, b, c). It is understood that the operations in the kth column are performed (mod m_k). For instance, if the ith column of the matrix consists entirely of zeros, we can divide - in the sense of solving linear congruences - all other elements of the matrix by the ith modulus and delete the ith column entirely. We thereby obtain a reduced set of residues which preserve the orientation of the cycle (a, b, c). We can also add or subtract a 1 to all elements of a row, which will either make two rows identical - in which case we will know whether (a, b, c) was positive or negative - or preserve the orientation of the cycle.

Clearly, when successive additions will not make 2 rows identical, these additions can be performed in one step. Thus, we can either determine the orientation of (a, b, c) by making two rows of the matrix become equal, or we can successively delete columns of the matrix until we have only one column remaining. In the latter case, the ordering of the residues in the last column is the same as the orientation of (a, b, c). Thus, if we know initially that $a < c$, we can use this procedure to obtain information about the magnitude of an "unknown" b.

As an example, let

$m_1 = 2$, $m_2 = 3$, $m_3 = 5$,

and

$m_4 = 7$ be the moduli.

We wish to test whether $18 < (1, 2, 4, 0) < 87$; that is, whether or not the cycle (a, b, c) is positive,

where

$a = (0, 0, 3, 4,)$,

$b = (1, 2, 4, 0)$,

and

$c = (1, 0, 2, 3)$,

using "positive residues." The procedure is as follows:

$$\begin{matrix} 0 & 0 & 3 & 4 \\ 1 & 2 & 4 & 0 \\ 1 & 0 & 2 & 3 \end{matrix} \xrightarrow[\text{2 and 3}]{}{}_{\!\!\!\!\!\!\text{add 1 to rows}} \begin{matrix} 0 & 0 & 3 & 4 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 3 & 4 \end{matrix} \xrightarrow[\substack{\text{divide by 2}\\ \text{and delete}\\ \text{1st column}}]{} \begin{matrix} 0 & 4 & 2 \\ 0 & 0 & 4 \\ 2 & 4 & 2 \end{matrix}$$

$$\xrightarrow[\substack{\text{add 1 to}\\ \text{row 3}}]{} \begin{matrix} 0 & 4 & 2 \\ 0 & 0 & 4 \\ 0 & 0 & 3 \end{matrix} \xrightarrow[\substack{\text{divide by 3}\\ \text{and delete}\\ \text{1st column}}]{} \begin{matrix} 3 & 3 \\ 0 & 6 \\ 0 & 1 \end{matrix} \xrightarrow[\substack{\text{add 1 to}\\ \text{row 1}}]{} \begin{matrix} 4 & 4 \\ 0 & 6 \\ 0 & 1 \end{matrix} \xrightarrow[\substack{\text{add 1 to}\\ \text{row 1}}]{}$$

$$\begin{matrix} 0 & 5 \\ 0 & 6 \\ 0 & 1 \end{matrix} \xrightarrow[\substack{\text{divide by 5}\\ \text{and delete}\\ \text{1st column}}]{} \begin{matrix} 1 \\ 4 \\ 3 \end{matrix}$$

Since (1, 4, 3) is a negative cycle, we conclude that (a, b, c) is negative; that is, b is not between 18 and 87. (Actually, b is 119.)

It should be noted that this technique is actually a generalization of the technique given in paragraph 4. 1. 1 for converting a number from residue form to two-sided mixed radix notation. In fact, if we let $a = 0$ and $c = M/2$, then the above technique will convert b to two-sided mixed radix.

Since there is a possibility that this technique will result in an improved magnitude determination algorithm and a good probability that it can be used for ordering large sets of numbers (an important operation in many data processing applications), further study of cycles and their application is recommended.

4. 3. 3 Table Look-up Procedures Using Mixed Radix Notation

A study of efficient table look-up procedures for modular arithmetic is recommended since several important algorithms require table look-up.

Since the mixed radix notation provides a useful "ordering" of the integers in a modular arithmetic system, it is natural to use some of the special properties of the mixed radix coefficients to speed the look-up procedure. For example, in using the table of powers of two in the division, square-root, and floating point procedures described above, the "entry point" in the table could be made to depend on the index and value of the highest order non-zero mixed radix coefficient of the number whose "binary logarithm" is being sought. If all but the two "low-order" mixed radix coefficients of an integer were 0, say, then we would begin our table look-up near the bottom of the table.

This study is related to the study of modular addressing techniques discussed in paragraph 6. 5. 2, and the two studies should be performed simultaneously.

5. SIMULATION OF MODULAR ARITHMETIC COMPUTERS ON THE IBM 7090

5.1 GENERAL

When simulating one computer (the source computer) on another (the host computer), it is customary to represent selected special registers (such as the accumulator and quotient registers) of the source computer by certain specific memory locations or registers of the host computer. Moreover, specific types of memories of the source computer (such as operand memory and instruction memory) are represented by blocks of memory of comparable size on the host machine. Finally, of course, each instruction or order code on the source computer is represented by an appropriate set of instructions on the host computer which accomplishes "the same" results. The results on the host computer are judged to be "the same" as those on the source computer if the operands are operated upon in such a way as to produce the same contents for the appropriate simulated registers as would be produced in the corresponding registers of the source computer after actual execution of the instruction. A simulation model containing these basic features is considered to be the classical computer simulation model.

5.2 DESCRIPTION

The model developed for the simulation of modular arithmetic (MA) computers on the IBM 7090 is not the classical one, although this may become desirable at some future date. Basically, the MA Simulator departs from the classical model in that no specific instruction repertoire is assumed for the MA computer. Rather, the simulation is of functions instead of instructions.

The MA Simulator can simulate any modular arithmetic computer configuration of from 2 to 12 moduli (m_i) where all moduli are less than 64 and at least two moduli are less than 32. The reasons for these restrictions will be discussed below; however, they are not too severe since the first 12 primes produce a range, M, in excess of 10^{11}, which is deemed ample for most feasibility investigations. The MA computer being simulated may have the following functions:

a. Addition

b. Subtraction

c. Multiplication

d. Division (solution of a linear congruence)

e. Division (iteratively by the method of paragraph 4.1.5.4)

f. Equal, not-equal compare

g. Compare magnitude

h. Sign determination

i. Conversion (decimal-to-modular)

j. Conversion (modular-to-decimal)

k. Conversion (modular-to-two-sided mixed radix)

l. Input (magnetic tape)

m. Output (magnetic tape, cards, or on-line printer)

For input/output purposes, the range, $CAPM = \prod m_i$, is considered to extend over the interval $(-\frac{CAPM}{2}, +\frac{CAPM}{2})$, approximately. To be sure, an MA computer would also have some other things like an instruction memory, data memory, and perhaps, index registers. But these are not the items being studied by the Model 1 Simulator. Effectively, only residue class arithmetic is being simulated.

Any applications problem involving only the functions listed above can be run on the available Model 1 Simulator. In fact, the test problems used to check out the simulator involved matrix addition, subtraction, and multiplication. The applications program is written in the FORTRAN language and, therefore, all of the FORTRAN input/output features are available to the MA programmer. Moreover, the applications program is compiled by the FORTRAN compiler and the simulator is run on the IBM 7090 as a normal FORTRAN operation.

5.3 METHOD

The primary purpose of the simulator was to assist in the investigation and evaluation of various numerical techniques. It was felt that this purpose would be greatly facilitated by the capability to write the applications programs in a problem oriented language with all of the capabilities of FORTRAN. After considering numerous possibilities, an ingenious method was found which permits the MA applications programs to be written in FORTRAN and compiled by the existing standard FORTRAN compiler.

The simulator was completed with a minimum of effort by taking advantage of the method FORTRAN uses to implement double-precision arithmetic. In the IBM 7090 FORTRAN II language, the appearance of the letter D in card column one of an arithmetic statement causes it to be compiled as a double-precision arithmetic statement. Among other things this means that the compiler will

a. allocate two words of storage to each double-precision variable

b. set up linkages to one of several unique subroutines which implement the basic operations of addition, subtraction, multiplication, and division.

Modular arithmetic can be forced into the framework of double-precision arithmetic by replacing the subroutines for the double-precision arithmetic operations with subroutines for their modular counterparts and by restricting the number and size of the moduli to allow packing the entire n-tuplet into two words. In this way, the existing FORTRAN compiler can process modular arithmetic programs. The replacement of the basic arithmetic subroutines by their modular counterparts does not involve any modification to FORTRAN since this capability is available in the standard FORTRAN system.

Any configuration of moduli which can be packed into 70 bits could have been used. However, since it was unlikely that the need would arise for moduli of magnitude greater than 64, it was expedient to standardize on a fixed format of six bits for each modulus. Since the sign bits of the two words cannot easily be used as magnitude bits, two of the moduli must be restricted to five bits.

The functions listed in (d) through (k) are effected through the use of FAP-coded subroutines which are "CALLable" in the applications program by standard FORTRAN CALL statements. In converting from decimal to modular form, the decimal number is divided by each of the moduli and the remainders are stored. The modular-to-decimal conversion routine makes use of the Chinese Remainder Theorem.

With the addition of each new feature to the simulator, greater and greater care had to be exercised to ensure compatibility with the earlier features. For example, prior to the addition of the features using the two-sided mixed radix representation, a least nonnegative residue representation was required. However, no conflicts arose through the use of both representations.

5.4 RESULTS

In general, work on the Modular Arithmetic Simulator program proceeded in a fashion typical of many research and development endeavors. That is, the course of the development of the simulator was dictated by the results of other investigations. Quite early in the development of the simulator (when the only method programmed for division was by solution of linear congruences), a matrix inversion application program was successfully run on the simulator. A sample of the actual results from the program is included as figures 5-1, 5-2, and 5-3. The 10 by 10 matrix to be inverted and its modular inverse are shown in figures 5-1 and 5-2 respectively. To facilitate verifying the results, the program exhibits the product of the matrix and its computed inverse, as shown in figure 5-3. For this sample, an MA computer having a range of -681,891 to +681,891 as produced by the four moduli 29, 31, 37, and 41 was simulated. All of the numbers in the output must be interpreted modulo the product of the moduli (1,363,783). For example, the element in the last row and last column of figure 5-2 represents the decimal number 2/4 which is -681,891. That is, -681,891 is a solution to the linear congruence $4x \equiv 2 \pmod{1,363,783}$.

Work on the implementation of one method for computing the adjoint of quasi-singular matrices was abandoned when a much simpler method was discovered. An applications program employing this improved method was prepared for the simulator and some results from the program appear in figures 5-4 and 5-5. Figure 5-4a is a 3 by 3 matrix whose determinant is -6. Using as moduli the numbers 3, 5, and 7, the program yields the matrix in figure 5-4a as the adjoint. To facilitate verifying the results, the program exhibits the product of the matrix and its computed adjoint as shown in figure 5-4c. Figures 5-5a, b, and c show the corresponding results from the program for a larger matrix for which the inverse is known explicitly. The numbers 2, 3, 13, 17, 29, and 41 were used as moduli.

In keeping with the intent that the simulator be used as a dynamic tool, the last applications program prepared for it was designed to answer empirically the question of number of bits per iteration of the Divide algorithm. The unique thing about this particular applications program is that it was not wholly modular. That is, the program may be thought of as having been run on two computers - a modular arithmetic computer in conjunction with a conventional computer. First, the conventional part of the program is used to randomly choose two numbers which are used by the modular part of the program as dividend and divisor in the Division algorithm discussed in paragraph 4.1.5.4 of this report. Following the division, another "conventional" part of the program is used to compute all of the quantities (other than the quotient) listed in the sample of the output shown in figure 5-6. The program repeats this process any desired number of times. From figure 5-6 it can be seen that after 500 divisions the average number of bits per iteration is 2.85. Because of space limitations, only the last page of the results is included herein.

NR = 10 NC = 10									
10.0	9.0	8.0	7.0	6.0	5.0	4.0	3.0	2.0	1.0
9.0	9.0	8.0	7.0	6.0	5.0	4.0	3.0	2.0	1.0
8.0	8.0	8.0	7.0	6.0	5.0	4.0	3.0	2.0	1.0
7.0	7.0	7.0	7.0	6.0	5.0	4.0	3.0	2.0	1.0
6.0	6.0	6.0	6.0	6.0	5.0	4.0	3.0	2.0	1.0
5.0	5.0	5.0	5.0	5.0	5.0	4.0	3.0	2.0	1.0
4.0	4.0	4.0	4.0	4.0	4.0	4.0	3.0	2.0	1.0
3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	2.0	1.0
2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	1.0
4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0

Figure 5-1. The A Matrix

THE INVERSE MATRIX (10 X 10)									
1.0	-1.0	0.	0.	0.	0.	0.	0.	0.	0.
-1.0	2.0	-1.0	0.	0.	0.	0.	0.	0.	0.
0.	-1.0	2.0	-1.0	0.	0.	0.	0.	0.	0.
0.	0.	-1.0	2.0	-1.0	0.	0.	0.	0.	0.
0.	0.	0.	-1.0	2.0	-1.0	0.	0.	0.	0.
0.	0.	0.	0.	-1.0	2.0	-1.0	0.	0.	0.
0.	0.	0.	0.	0.	-1.0	2.0	-1.0	0.	0.
0.	0.	0.	0.	0.	0.	-1.0	2.0	-1.0	0.
0.	0.	0.	0.	0.	0.	0.	-1.0	2.0	-340946.0
0.	0.	0.	0.	0.	0.	0.	0.	-1.0	-681891.0

Figure 5-2. The Inverse Matrix

1.0	0.	0.	0.	0.	0.	0.	0.	0.	0.
0.	1.0	0.	0.	0.	0.	0.	0.	0.	0.
0.	0.	1.0	0.	0.	0.	0.	0.	0.	0.
0.	0.	0.	1.0	0.	0.	0.	0.	0.	0.
0.	0.	0.	0.	1.0	0.	0.	0.	0.	0.
0.	0.	0.	0.	0.	1.0	0.	0.	0.	0.
0.	0.	0.	0.	0.	0.	1.0	0.	0.	0.
0.	0.	0.	0.	0.	0.	0.	1.0	0.	0.
0.	0.	0.	0.	0.	0.	0.	0.	1.0	0.
0.	0.	0.	0.	0.	0.	0.	0.	0.	1.0
FINISHED									

Figure 5-3. Product of A and A Inverse

NR = 3 NC = 3		
The A Matrix	The Adjoint Matrix (3 x 3)	Product of A and A Adjoint
1.0 1.0 0.	-6.0 -1.0 1.0	-6.0 0. 0.
1.0 -1.0 1.0	0. 1.0 -1.0	0. -6.0 0.
1.0 5.0 1.0	6.0 -4.0 -2.0	0. 0. -6.0
		FINISHED
a.	b.	c.

Figure 5-4. Results of Improved Method of Computing The Adjoint of Quasi-Singular Matrices

THE A MATRIX						
a			NR = 7 NC = 7			
3.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	3.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	3.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	3.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	3.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	3.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	3.0
b			THE ADJOINT MATRIX (7 X 7)			
256.0	-32.0	-32.0	-32.0	-32.0	-32.0	-32.0
-32.0	256.0	-32.0	-32.0	-32.0	-32.0	-32.0
-32.0	-32.0	256.0	-32.0	-32.0	-32.0	-32.0
-32.0	-32.0	-32.0	256.0	-32.0	-32.0	-32.0
-32.0	-32.0	-32.0	-32.0	256.0	-32.0	-32.0
-32.0	-32.0	-32.0	-32.0	-32.0	256.0	-32.0
-32.0	-32.0	-32.0	-32.0	-32.0	-32.0	256.0
c			PRODUCT OF A AND A ADJOINT			
576.0	0.	0.	0.	0.	0.	0.
0.	576.0	0.	0.	0.	0.	0.
0.	0.	576.0	0.	0.	0.	0.
0.	0.	0.	576.0	0.	0.	0.
0.	0.	0.	0.	576.0	0.	0.
0.	0.	0.	0.	0.	576.0	0.
0.	0.	0.	0.	0.	0.	576.0
FINISHED						

Figure 5-5. Results of Improved Method of Computing
The Adjoint of Quasi-Singular Matrices
(Inverse Known Explicitly)

	DIVIDEND	DIVISOR	QUOTIENT	J	K	D	ALPHA	MEAN
482	243186.	612294.	832930.	-21	10	24	2.40	2.84
483	204951.	199553.	538470.	-19	7	22	3.14	2.84
484	440759.	85629.	674668.	-17	8	21	2.63	2.84
485	789684.	499616.	828680.	-19	10	23	2.30	2.84
486	720839.	773454.	977246.	-20	8	22	2.75	2.84
487	528507.	473383.	585340.	-19	7	22	3.14	2.84
488	231992.	102234.	594864.	-18	6	22	3.67	2.84
489	226482.	843953.	562789.	-21	9	20	2.22	2.84
490	314317.	781404.	843572.	-21	8	25	3.13	2.84
491	546160.	493487.	580249.	-19	13	21	1.62	2.84
492	845038.	741395.	597580.	-19	7	21	3.00	2.84
493	278702	582430.	1003520.	-21	4	21	5.25	2.84
494	145388.	50191.	759351.	-18	9	22	2.44	2.84
495	232847.	587848.	830683.	-21	12	22	1.83	2.84
496	309945.	308665.	526462.	-19	4	22	5.50	2.85
497	137488.	494436.	583156.	-21	8	22	2.75	2.85
498	638371.	743763.	899992.	-20	7	22	3.14	2.85
499	9624.	375516.	859954.	-25	9	21	2.33	2.85
500	317891.	199362.	835999.	-19	8	25	3.13	2.85
FINISHED								

Figure 5-6. Sample Output

6. SYSTEMS AND LOGIC DESIGN

6.1 INTRODUCTION

The range of immediate applications for modular arithmetic computers appears to be limited to something less than the full spectrum of applications to which conventional digital computers are applied. This statement assumes that no dramatic improvement will be made in present modular arithmetic algorithms, specifically in sign and magnitude determinations, division, and floating-point operations.

The potential application area for modular digital computers is that presently occupied by conventional fixed-point computers used for the solution of scientific problems (as opposed to data processing). In addition, it seems likely that modular computers will be able to solve special problems which heretofore have been impossible to solve because of the large numbers of fast multiplications required. For example, it is entirely feasible and practical to implement with semiconductor devices a 10-mc modular multiplier with any reasonable word length (roughly, 60 bits or less).

Because of our recent advances in modular adder and multiplier mechanizations, it is now feasible and economical to implement both parallel and serial modular fixed-point computers for both small and large machine ranges. Thus, while further studies will no doubt yield further economies in implementation, the present status of modular arithmetic algorithms and implementation techniques is such that modular arithmetic now warrants careful consideration for application to most fixed-point computer problems. The systems and logic design sections to follow are of an adequately general nature to be applied to the design of almost any modular computer.

6.2 MODULAR COMPUTER SYSTEMS ORGANIZATION

For most applications, a modular computer would be identical to an ordinary fixed-point digital computer in its basic organization. The modular system to be described below, if carried to a detailed design for a particular application, would provide capabilities conceptually similar to those of a fixed-point computer. In the remainder of this section, the major units of the computer are described functionally; in the following section, the arithmetic unit and the control unit are described in more detail.

6.2.1 Input Unit

This unit will contain various input equipment as required, depending on the computer application. All incoming quantities can be stored in binary registers. These registers will store fixed radix quantities only, which will be routed to the modular arithmetic unit for conversion to residue code, after which the residues will be stored in memory. For all practical purposes the input unit will be organized precisely in some conventional form and need not be described further here.

6.2.2 Memory Unit

The storage requirements of the modular computer will not differ greatly from those of fixed radix computers. In almost all memory retrievals, the memory will be instructed to retrieve some one set of residues; i.e., one operand, needed for the computer operation specified by control. Therefore, it seems natural to organize the memory word in the following fashion. The residues will be stored in least absolute value residue form for each of the system moduli $m_1, m_2, \ldots, m_n$. Each residue will be coded in binary notation, with modulus m_1 requiring b_1 bits, where b_1 is the first integer $\geq \log_2 m_1$. The memory word which stores quantity $x = (x_1, x_2, \ldots, x_n)$, $x_i = |x|_{m_i}$, will contain residue x_1 in bit positions 1, 2, ..., b_1; residue x_2 in bit positions $b_1 + 1$, $b_1 + 2$, ..., $b_1 + b_2$; and similarly for the other residues. The leftmost bit for each modulus will represent the sign bit.

An estimate of storage efficiency as compared to a binary computer memory can be obtained from an example. Consider a conventional computer with number range of $10^8 \cong 2^{27}$. Neglecting sign and parity bits, such a computer would require 27-bit memory registers. A modular computer with a similar number range and for the set of moduli 31, 32, 51, 53, and 55, would require, in order of increasing moduli, $5 + 5 + 6 + 6 + 6 = 28$ bits for each memory register. Thus for this set of system moduli, the modular computer would require about 4 percent more storage capacity than the fixed radix computer for the same machine range. Obviously, storage efficiency is dependent upon the efficiency for each modulus. Proper choice of moduli for any system; i.e., a choice weighing minimum adder-multiplier hardware against maximum storage efficiency, will never require more than 10 percent additional memory for the modular machine.

6.2.3 Control Unit

This unit of the modular computer will contain the clock and the necessary control logic to direct and coordinate the various functions of the machine.

Paragraph 6.3.2 gives details of a "typical" control unit design. One possible feature of a modular control unit not treated in paragraph 6.3.2 is that of a special control requirement encountered in matrix inversions or solutions of sets of linear equations. If a modular computer is required to solve these problems, then all the moduli must be primes in order that solutions to the linear congruences which result will always exist. Also, control logic must be provided to facilitate the interchanging in 2 memory locations of the residues for a particular modulus, while leaving the remainders of these memory words unaltered. This is because elements of a matrix can be congruent to 0 modulo m_i, but, not congruent to 0 modulo m_j. At some point in the matrix inversion procedures, elements congruent to 0 mod m_i have to be interchanged for nonzero mod m_i elements.

6.2.4 Arithmetic Unit

Given a modular computer with n moduli, the arithmetic unit of the machine will consist of n modular adders, n modular multipliers, an accumulator with residues organized in the same fashion as in a memory register, and some routing logic. As will be described below, the adders and multipliers are the heart of the computer, functioning for input conversion, all arithmetic operations, and output conversion. In the present design, the computer is organized as a single-address machine in which, of any two modular quantities to be operated on, one is always taken from the accumulator and the other from some memory register (or also from the accumulator in special cases), with the result routed back to the accumulator. These generalities will be elaborated upon in paragraph 6.3.1.

6.2.5 Output Unit

If the modular-to-fixed radix conversion procedure described in paragraph 4.1.4 is used, the only hardware required in the output unit for conversion will be a register or registers for holding fixed radix quantities arriving from the accumulator or memory after a conversion has been made. If the method of conversion of paragraph 4.1.4 is too slow for some special high output rate application, a faster method can be obtained by solving the Chinese Remainder Theorem problem in the fashion outlined below.

The Chinese Remainder Theorem states that,

given

$$x \equiv x_1, x_2, \ldots, x_n \mod (m_1, m_2, \ldots, m_n),$$

where the moduli are pairwise prime, then

$$x \equiv c_1x_1 + c_2x_2 + \ldots + c_nx_n \pmod{M},$$

where the c_i are constants which may be calculated in advance and stored to be retrieved when needed. Previous solutions have assumed a requirement for fixed radix multiplication to obtain the products c_ix_i and fixed radix addition to sum these products. Further, to obtain rapid solutions, parallel addition and parallel multiplication were required. The hardware for conversion was thus comparable to that of an entire arithmetic unit of a fixed radix parallel machine. This approach obviously forfeits much of the advantage of modular arithmetic.

A better solution to this problem follows simply from the observation that the quantities c_ix_i do not have to be obtained by multiplication. Since there are only m_i possible values of the quantity c_ix_i, (i.e., 0, $1c_i$, ..., $(m_i-1)c_i$), it is entirely feasible to store all possible c_ix_i for each m_i in a read-only memory and reduce the problem to a series of memory retrievals and additions. Consider, for example, a modular computer employing as moduli the primes 2 through 31, giving a machine number range of about 10^{11}. The suggested approach requires the storage of 150 predetermined constants of around 40 bits each, or a total requirement of 6000 bits. A parallel adder is still required, but the hardware requirement is much less than that when parallel multiplication is provided. Further, the solution will be obtained many times faster, depending on the number of moduli used.

In summary, the reconversion process requires a small read-only memory, a parallel adder, and the necessary control logic. Assuming the parallel adder adds in one clock time, the reconversion for a system using n moduli will require either n memory cycles or n clock times, whichever is greater.

6.3 ARITHMETIC AND CONTROL UNITS

The typical modular arithmetic and control units to be described in this paragraph assume a reasonable minimum instruction set. Thus, for example, the divide operation is performed with a subroutine, as is the modular-to-fixed-radix conversion operation.

6.3.1 <u>Arithmetic Unit</u>

The arithmetic unit with the proper control can execute the following instructions:

1. Addition - ADD
2. Subtraction - SUB
3. Multiplication - MUL

4. Convert Modular to TSMR - CON
5. Compare Magnitude of x and y - COM
6. Transfer on $x > y$ - TRA
7. Convert Modular to TSMR, determine sign, transfer if sign is positive - DES
8. Load the accumulator - LOD
9. Store - STO
10. Convert binary input to modular - INP
11. Output - OUT
12. Circular Shift - CIR
13. Logical AND - AND
14. Logical OR - LOR

A description of how each of the above operations is accomplished will now be given.

1. Addition

Addition (modular, of course) in the arithmetic unit is straightforward. Given a number $x = (x_1, x_2, \ldots, x_n)$ in the accumulator and the address of a number $y = (y_1, y_2, \ldots, y_n)$ in memory and instruction from control to add $z = x + y$; (1) y is retrieved from memory, (2) x and y are fed to the inputs of the adders, and (3) z is routed back and stored in the accumulator.

2. Subtraction

Since the operands are stored in LAVR code, to subtract $x - y$, complement the sign bit of each $(y_1, y_2, \ldots, y_n)$ and add as above.

3. Multiplication

Given a number $x = (x_1, x_2, \ldots, x_n)$ in the accumulator and the address of a number $y = (y_1, y_2, \ldots, y_n)$ in memory, and instruction from control to multiply $x \cdot y = z$; (1) y is retrieved from memory, (2) x and y are fed to the inputs of the multipliers, and (3) z is stored in the accumulator.

4. Modular to TSMR Conversion

Given a number $x \equiv (x_1, x_2, \ldots, x_n)$ in least absolute value residue notation for which TSMR conversion is desired; (1) the subtractions $x_i - x_1$ are performed for all $i > 1$, (2) the modular inverses d_{1i}, $(i = 2, \ldots, n)$, are retrieved from memory, and (3) the products $(x_i - x_1) \cdot d_{1i} \pmod{m_i}$, are formed for all $i > 1$. The result of these three steps is the elimination of modulus m_1; a similar set of operations removes modulus m_2, etc, until only the one modulus, m_n, remains. The TSMR number is then

$$x = a_1 + a_2(m_1) + a_3(m_1 m_2) + \ldots + a_n (m_1 m_2 \ldots m_{n-1}).$$

The time required for a TSMR conversion is a function of the number of moduli; namely, if there are n moduli in the residue system, the time required is 2(n - 1) clock times, allowing one clock time each for (1) a memory retrieval and a subtraction and (2) a multiplication. This time can be reduced to (n-1) clock times, if multiplication by the constants d_{ij} is accomplished through decoding binary residues to a one-of-m is hot code, wire-twisting to give the proper multiplication by d_{ij}, and encoding to binary residues. This is, in general, relatively expensive of hardware, requiring, for example, on the order of 500 logic gates for system moduli 32, 51, 53, and 55.

5. Compare Magnitude of x and y

Two operands x and y to be compared must be available in TSMR form. If the signs of their rightmost TSMR coefficients a_n and b_n differ, then the signs of x and y differ and the comparison is completed. If a_n and b_n are of like sign, they are compared algebraically. If both are positive, then the larger of the two belongs to the larger number; if both are negative, the lesser negative belongs to the larger number. If $a_n = b_n$, the same test is applied to a_{n-1} and b_{n-1}, etc, until the comparison is completed.

6. Transfer Program on $x > y$

This is a conventional transfer instruction.

7. Convert Modular to TSMR, determine sign, transfer on positive

This instruction is an obvious extension of instruction (4). Further details will be given in paragraph 6.3.2.

8. Load the Accumulator

This instruction loads the accumulator with the contents of the memory location specified by the address field of the instruction.

9. Store

The contents of the accumulator are stored in the specified memory location.

10. Convert Binary Input to Modular

The iterative conversion procedure from binary notation to least absolute value residue form will be reviewed briefly here and then discussed in terms of how it is implemented in the computer.

Given a number x in binary code, let

$$x = 2^k a_o + 2^{k-1} a_1 + \ldots + 2 a_{k-1} + 2^0 a_k,$$

where

$$a_o = 1, \; a_i = 0 \text{ or } 1 \text{ for } i \neq 0.$$

Define the recurrence

$$x_{i+1} = 2 x_i + a_{i+1}.$$

Then,

$x_k = x$ and it follows that

$x_k \equiv x \pmod{m_j}$

for all system moduli m_j. The discussion below concerns what is necessary for obtaining the residue of x for each modulus; conversion will of course proceed simultaneously for each modulus.

We must calculate x_k by going through k iterations, where k is one less than the binary bits required to represent the number. Each iteration is a left shift by one bit and an addition of either 1 or 0 to the shifted quantity. Alternatively, we can add to the quantity x_i the quantity a_{i+1} and then multiply by 2, so conversion can be accomplished in 2k clock times. These operations are of course all modular and with respect to each modulus. Then, given a number in a register in the input unit to be converted, control will specify a set of k iterations as described above. If the input quantities are 24 bit numbers and a 1-mc clock is used, conversion will require 48 microseconds per 24-bit word.

The possibility exists of course for reducing the number of iterations and consequently the conversion time by expressing input quantities in higher radix form; e.g., 16. If hexadecimal code were used, there would be only one-fourth the iterations required in the binary-to-modular operation just described. Since the implementation methods of Section 7 make it economically feasible to mechanize adders and mult pliers for large moduli, it is quite possible that each modulus will be greater n 16. If this is the case, then the binary k-bit number can be operated on as a h decimal (k/4)-digit number. For k = 24 and a 1-mc clock, the hexadecimal conversion requires 12 microseconds. A minor complication arises if moduli less than 33 are used with LAVR, since the four-bit characters (hexadecimal digits) are not automatically the correct LAVR residues for those moduli. Obvious solutions are (1) make all moduli greater than 32, (2) use octal digits, which are correct LAVR residues for these moduli, and (3) use supplementary hardware to convert to correct LAVR residues for these moduli.

11. Output

The contents of the accumulator are placed in an output register. Accumulator contents can be either modular, mixed radix, or fixed radix, so the form of the output will depend on the program.

12, 13, 14. Left Circular Shift, Logical AND, Logical OR

These instructions have been included primarily for the modular-to-fixed radix conversion subroutine, but undoubtedly can be used elsewhere. (For example, the AND instruction can be used to reconstruct memory words as is required for matrix inversion.) Functionally, a CIR instruction shifts the contents of the accumulator a specified number of bits to the left with the overflow bits shifted into the low order bit positions. The AND instruction performs the logical AND of two words and stores the contents in the accumulator. The LOR instruction performs the logical OR of two words and stores the contents in the accumulator.

An example of the modular-to-fixed-radix conversion described in paragraph 4.1.4 will now be given.

Consider a system with moduli 2, 3, 5, 7, 11, and 13. A conversion from LAVR to decimal notation is to be made. Let

$x \equiv 0, -1, -2, -3, -5, 6 \pmod{2, 3, 5, 7, 11, 13}$.

The problem is to convert x to decimal notation using only modular operations. The first step is to get the residue of x modulo 10. This residue is the sum of the first two terms of the mixed radix representation.

$x = a_1 + a_2 (2) + a_3 (2 \cdot 5) + \ldots + a_n (2 \cdot 5 \cdot 3 \cdot 7 \cdot 11)$.

Now,

$a_1 = x_2 = 0,$

$a_2 = (x_5 - x_2) \cdot d_{25} = (-2 -0)(-2) = 4$

Thus,

$x_{10} = a_1 + a_2 \cdot 2 = 0 + 8 = 8$

The integer, 8, is then the units digit of x in decimal notation. To obtain the tens digit of x, subtract x_{10} from each residue of x and divide by 10; i.e., by moduli 2 and 5. Proceed as follows.

Let

$$N_1 \equiv \frac{x_2 - x_{10}}{2 \cdot 5}, \frac{x_5 - x_{10}}{2 \cdot 5}, \frac{x_3 - x_{10}}{2 \cdot 5}, \ldots, \frac{x_{13} - x_{10}}{2 \cdot 5}$$

$$\equiv \left|N_1\right|_2, \left|N_1\right|_5, 0, \frac{+3}{2 \cdot 5}, \frac{-2}{2 \cdot 5}, \frac{-2}{2 \cdot 5} \bmod (2, 5, 3, 7, 11, 13)$$

Note that $\left|N_1\right|_2$ and $\left|N_1\right|_5$ are not known. They must be obtained by applying the mixed radix conversion to N_1 until only the moduli 2 and 5 remain. At this point we have

$\left|N_1\right|_2 - 1 \equiv 0 \pmod 2$ $\qquad 2\left|N_1\right|_5 + 1 \equiv 0 \pmod 5$

$\left|N_1\right|_2 = 1$ $\qquad \left|N_1\right|_5 = 2$

The tens digit of x is then the sum of the first two mixed radix terms of N_1.

$a_1 = \left|N_1\right|_2 = 1$

$a_2 = (\left|N_1\right|_5 - \left|N_1\right|_2) d_{25} = (2-1)(3) = 3$

Thus,

$\left|N_1\right|_{10} = 1 + 3(2) = 7$

In a similar manner, the remaining decimal digits of x can be obtained. We get $x = 578$.

It should be noted that the particular order in which the moduli are eliminated in solving for the unknown residues, as for $\left|N_1\right|_2$ and $\left|N_1\right|_5$ above, requires that equipment be provided for converting residues modulo 10 to the correct residues modulo the modulus from which the residue mod 10 is subtracted.

The way in which conversion from LAVR to decimal code can be programmed is generally as follows. The number x to be converted to decimal form is converted to TSMR form and placed in memory. The residue mod 10 is calculated in POSITIVE RESIDUE form (i.e., the true units digit of x) and placed in the four least significant bit positions of the previously cleared accumulator, and then stored in a memory location. The number N_1 is calculated next and from N_1, the tens digit of x or $\left|N_1\right|_{10}$. $\left|N_1\right|_{10}$ is shifted to bit positions 5 through 8 of the accumulator, and the logical AND of $\left|N_1\right|_{10}$ and a memory word which is all zeros except for ones in bit

The integer, 8, is then the units digit of x in decimal notation. To obtain the tens digit of x, subtract x_{10} from each residue of x and divide by 10; i.e., by moduli 2 and 5. Proceed as follows.

Let

$$N_1 \equiv \frac{x_2 - x_{10}}{2 \cdot 5}, \frac{x_5 - x_{10}}{2 \cdot 5}, \frac{x_3 - x_{10}}{2 \cdot 5}, \ldots, \frac{x_{13} - x_{10}}{2 \cdot 5}$$

$$\equiv \left|N_1\right|_2, \left|N_1\right|_5, 0, \frac{+3}{2 \cdot 5}, \frac{-2}{2 \cdot 5}, \frac{-2}{2 \cdot 5} \bmod (2, 5, 3, 7, 11, 13)$$

Note that $\left|N_1\right|_2$ and $\left|N_1\right|_5$ are not known. They must be obtained by applying the mixed radix conversion to N_1 until only the moduli 2 and 5 remain. At this point we have

$\left|N_1\right|_2 - 1 \equiv 0 \pmod 2$ $\qquad 2\left|N_1\right|_5 + 1 \equiv 0 \pmod 5$

$\left|N_1\right|_2 = 1$ $\qquad \left|N_1\right|_5 = 2$

The tens digit of x is then the sum of the first two mixed radix terms of N_1.

$a_1 = \left|N_1\right|_2 = 1$

$a_2 = (\left|N_1\right|_5 - \left|N_1\right|_2) d_{25} = (2-1)(3) = 3$

Thus,

$\left|N_1\right|_{10} = 1 + 3(2) = 7$

In a similar manner, the remaining decimal digits of x can be obtained. We get $x = 578$.

It should be noted that the particular order in which the moduli are eliminated in solving for the unknown residues, as for $\left|N_1\right|_2$ and $\left|N_1\right|_5$ above, requires that equipment be provided for converting residues modulo 10 to the correct residues modulo the modulus from which the residue mod 10 is subtracted.

The way in which conversion from LAVR to decimal code can be programmed is generally as follows. The number x to be converted to decimal form is converted to TSMR form and placed in memory. The residue mod 10 is calculated in POSITIVE RESIDUE form (i.e., the true units digit of x) and placed in the four least significant bit positions of the previously cleared accumulator, and then stored in a memory location. The number N_1 is calculated next and from N_1, the tens digit of x or $\left|N_1\right|_{10}$. $\left|N_1\right|_{10}$ is shifted to bit positions 5 through 8 of the accumulator, and the logical AND of $\left|N_1\right|_{10}$ and a memory word which is all zeros except for ones in bit positions 5 through 8 is formed in the accumulator. Next, the logical OR of x_{10} in memory and $|N_1|_{10}$ in the accumulator is formed in the accumulator. In a similar fashion, the remaining digits of the binary coded decimal form of the number x are generated.

6.3.2 Control Unit

The computer control unit was studied in such detail as seemed desirable. The main goal was to show that those commands which were different in form in a modular computer from their counterparts in a conventional computer could be implemented without using too many components. A total component count of reasonable accuracy was sought.

Six moduli, 2, 3, 5, 7, 11, and 13, are used to give a machine range of 30,030. These moduli are not necessarily those which would be used in an actual computer, but they serve as an adequate example.

The chosen computer repertoire of commands includes those which are considered desirable in most modular computers, but does not include commands which would occur only according to need in a given application. The commands chosen for inclusion are given with their symbols below:

1. Addition - ADD
2. Subtraction - SUB
3. Multiplication - MUL
4. Conversion from modular to TSMR form - CON
5. Compare TSMR numbers for magnitude - COM
6. Transfer program of A > B - TRA
7. Convert from modular to TSMR form, determine sign, and transfer program if sign is positive - DES
8. Load - LOD
9. Store - STO
10. Input conversion from binary to modular - INP
11. Output - OUT
12. Left Circular Shift - CIR
13. Logical AND - AND
14. Logical OR - LOR

Commands number 4, 5, 6, and 7 are interrelated. There are two major reasons for converting from modular to TSMR form. These are to determine the sign of a number and to compare the magnitude of two numbers. In both cases, the result will determine whether or not a transfer of program is initiated. When the sign is to be determined, the conversion and conditional transfer can be part of the same command. Instructions (5) and (6) are separate commands because the machine organization chosen does not provide for storing the data address and the transfer address simultaneously.

Commands 12, 13, and 14 are provided mainly for the programming of output conversion by the method of paragraph 4.1.4.

The computer control unit has four registers and two counters. These are the instruction register, the B register, the A register, the memory address register, the instruction counter, and the cycle counter, as seen in figure 6-1. There is a third four-bit shift counter which is part of the instruction register and a fourth two-bit counter which is part of the memory address register.

Since the six moduli require a total of 17 bits, the typical memory data word will be of this length. The B register which receives data from memory and the A register which acts as the computer accumulator are examples. The instruction register was also chosen to be 17 bits. Allowing the necessary four bit field for instructions, this leaves a memory address field sufficient size to address 8192 memory words. The instruction counter and the memory address register are therefore each of 13 bits. The C counter controls the general timing of the computer and has at least four bits.

The general cycle that the computer follows is that the instruction is read from memory into the instruction register after the instruction address is transferred from the instruction counter into the memory address register. The data address is then transferred from the instruction register into the memory address register and the data word is read into the B register where it is used in the performance of the instruction.

The add instruction, ADD, is very straightforward and is performed by sending an add signal to all the modular adders (see figure 6-2).

The subtract command, SUB, is performed by first sending a complement signal to all the signs of the modular numbers in the B register, and then sending an add signal to all the adders. Multiplication, MUL, is performed by sending a multiply

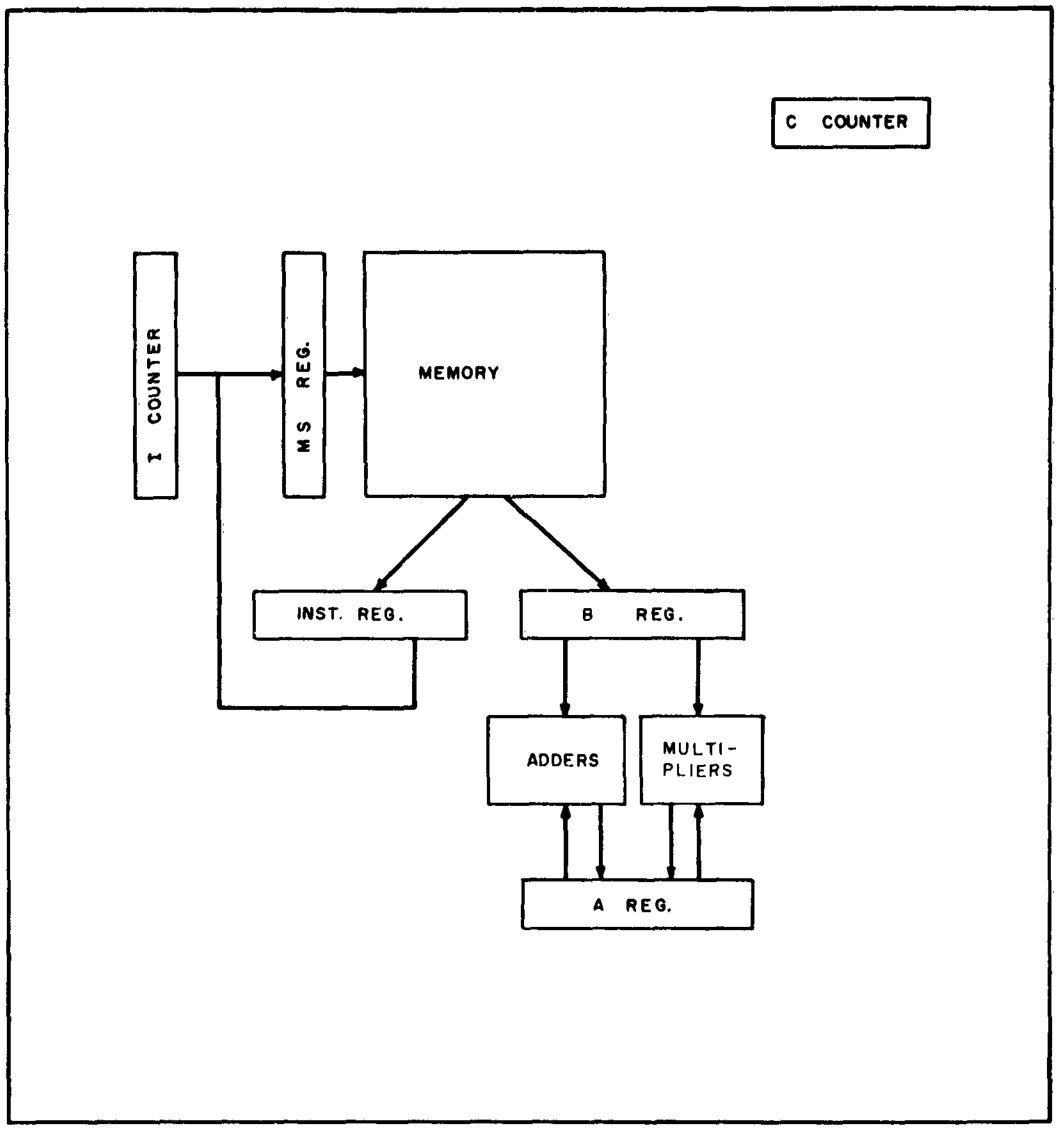


Figure 6-1. Modular Computer Organization

signal to all the modular multipliers. The results of add, subtract, or multiply end up in the A register.

The conversion from modular to TSMR command, CON, introduces somewhat more complexity. It is necessary to perform successive subtractions of various parts of the A register from other parts of the A register representing larger

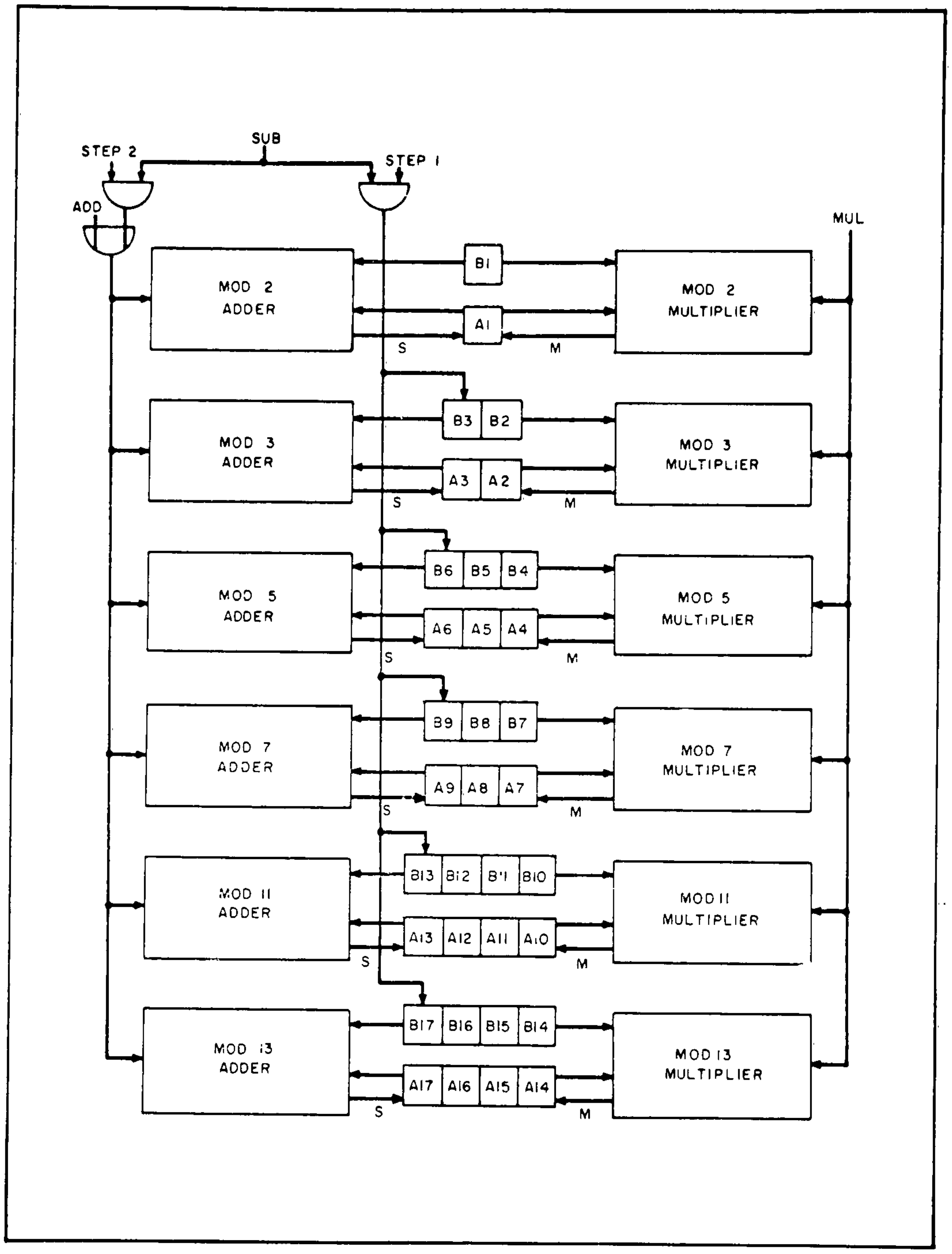


Figure 6-2. Arithmetic Control

moduli. This means, in effect, that there is a great deal of alternate gating to the adder inputs which receive from the B register during the command. This gating for the mod 13 adder, which is the worst case, is shown in figure 6-3.

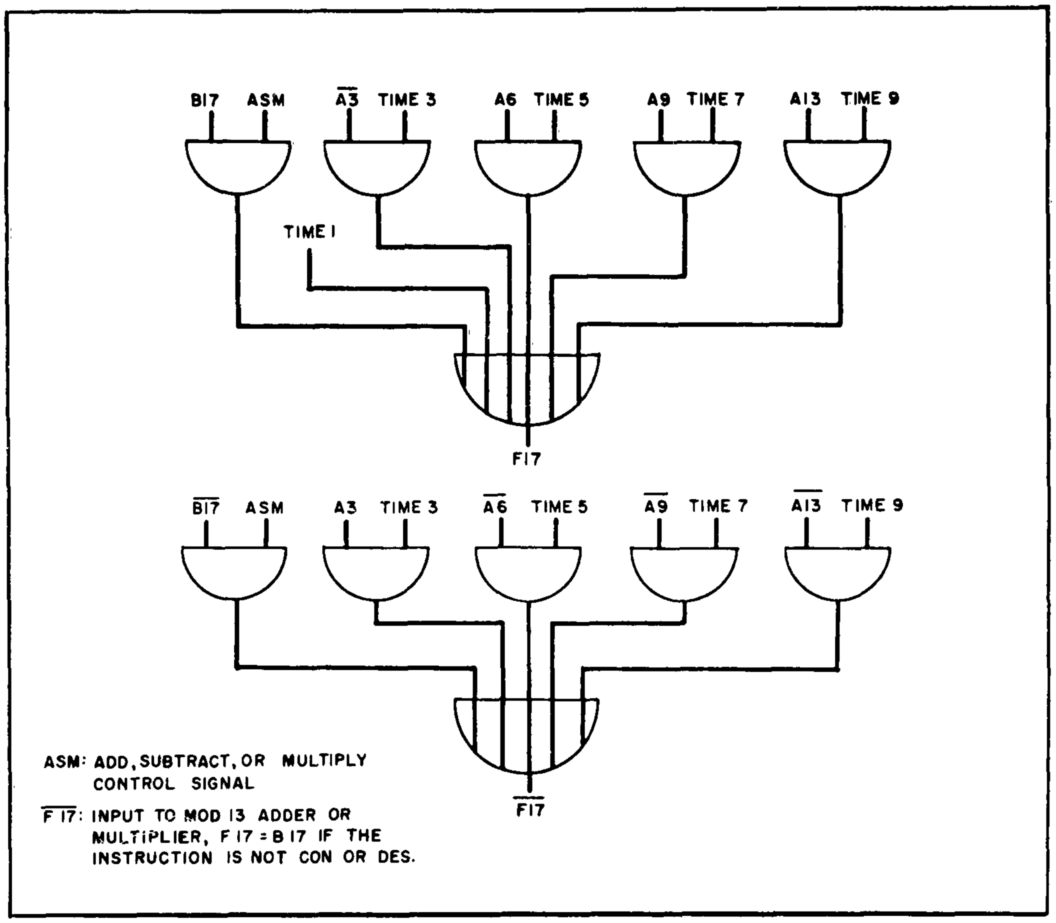


Figure 6-3. Representative Gating for TSMR Conversion

Since some moduli require a different number of bits than others and the adder being used in each case is that of the larger modulus, some of the adder inputs are filled in on the complement side. The other procedure necessary for this operation is to alternate multiplication by the various d_{kj} with the subtractions. This means that repeated memory access is performed during the time of the conversion command and it is necessary to provide means for variable addressing. This is done by forming a counter out of the three least significant bits of the memory address register. Each time a new data word is to be read from memory in the CON command, the counter is advanced by one. Certain trivial programming restrictions are caused as a result. In order that the counter may not overflow, it is necessary for the programmer to choose an initial address with the three least significant bits representing a small enough number.

The comparison command, COM, involves a chain of logic. The logic for the modulus 13 is shown in figure 6-4. The signs of the mod 13 numbers, a_n and b_n, are compared. If the sign of a_n, A17, is positive and the sign of b_n, B17, is negative, A is larger than B. If the two signs are the same, then the numbers a_n and b_n are compared step by step in the comparator shown in figure 6-5, starting with the most significant bits. If $a_n > b_n$ and both signs are positive or if $b_n > a_n$ and both signs are negative, then A is larger than B. If $a_n = b_n$, it is necessary to examine the signs and numbers of the next largest modulus in a similar manner. In this way, a sufficient number of moduli are examined to determine if A is larger than B and to set a flip-flop S if it is. As this command is comparatively expensive in hardware, the possibility of programming it should be considered.

The command to transfer the program if A > B, TRA, is used to follow a compare magnitude command. It examines the state of the flip-flop determined in the compare command. If A > B, the address in the instruction register is placed in the instruction counter.

The composite command, convert from modular to TSMR form, determine sign, and transfer program if sign is positive, DES, could consist only of determine sign since separate commands are available to perform the other functions. However, for the sake of economizing on memory time and since the facilities used for the separate commands are available for use in the complex command with very little extra control, it was decided to group these functions. Figure 6-6 shows the logic involved in determining the sign of the TSMR number. The S flip-flop is set (1) if a_n, the TSMR coefficient associated with modulus 13, is not zero and its sign is negative, or (2) if a_n is zero and a_{n-1} is not zero and its sign is negative, or (3) if the same conditions are met for any of the other moduli. If the sign is positive, the address in the instruction register is placed in the instruction counter.

The load command, LOD, loads the A register with a word from memory. This is done by clearing the A register, dumping a word from memory in the B register, and adding it to the A register.

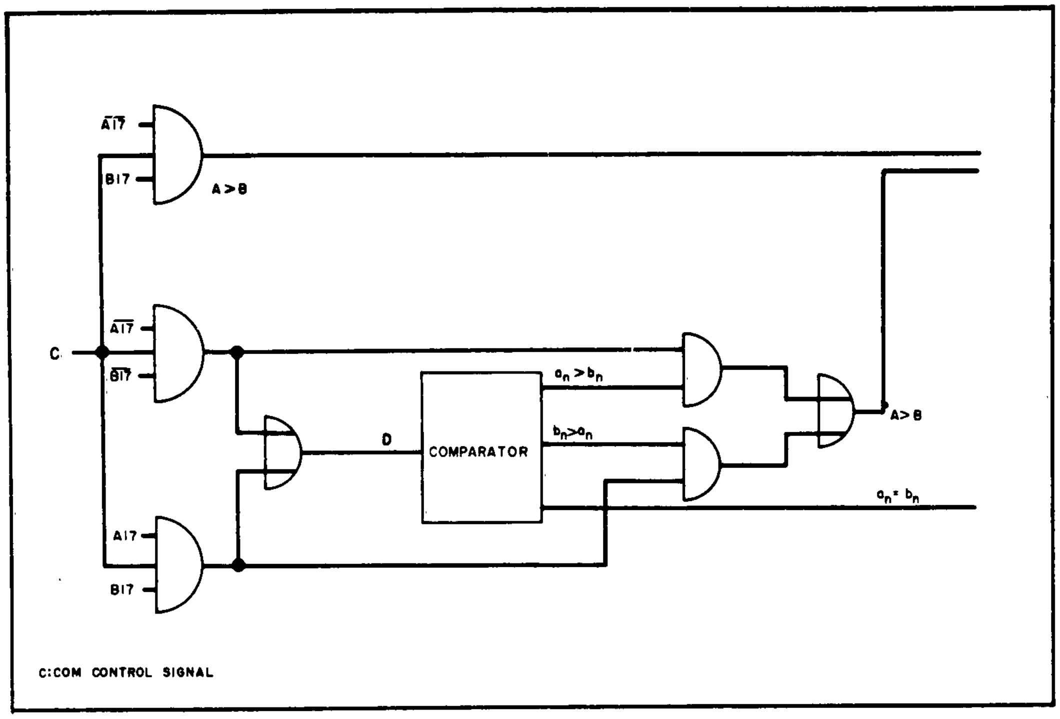


Figure 6-4. Comparison Command Logic

The store command, STO, operates by transferring a word from the A register to the B register while inhibiting the reading of the word in memory into B. During the write cycle, the word in B is stored in memory.

The input conversion from binary to modular command, INP, takes a word from the input unit in binary form and converts it into modular form in the A register. The A register is initially cleared and the binary word is input serially, most significant bit first. The first bit is entered into all the modular adders simultaneously on the B register adder inputs at the least significant position. The result is then multiplied by the number 2 located in the B register. The second most significant bit is then added to the number in the A register. The process is repeated with alternate additions and multiplications until all bits have been processed. The output command, OUT, simply takes the contents of the A register and transfers it to some register in the output unit where it is needed. The details will depend on the function of the computer.

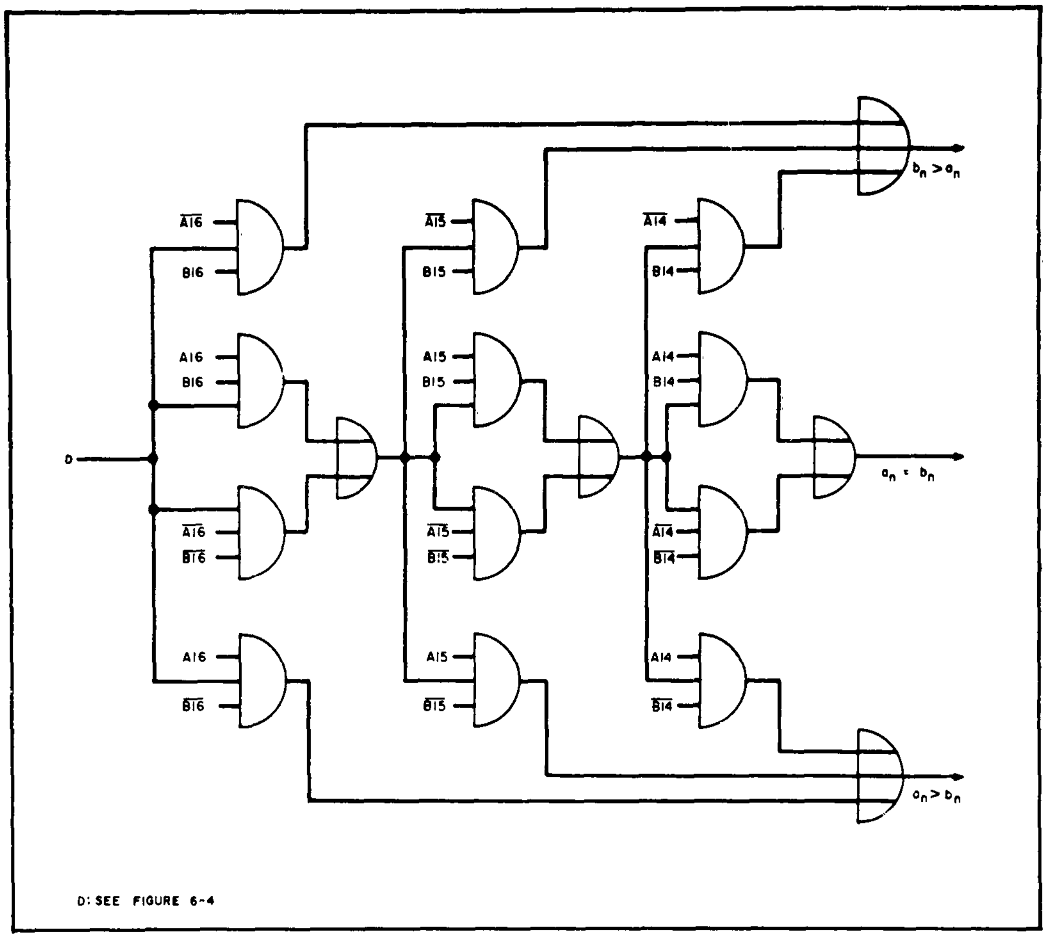


Figure 6-5. Comparator

The logical And command, AND, is implemented with an "and" gate for each bit position. The inputs to these gates, aside from control, are the corresponding bits of the A and B registers. The outputs of these gates go to the A register.

The logical Or command, LOR, is implemented in a way similar to that of the logical And except that the "and" gates are replaced by "or" gates and the control must be applied to "and" gates on the output of the "or" gates.

The circular shift command, CIR, is implemented by making a shift register out of the A register with each bit capable of shifting from 1 to 16 places to the left. The shift is circular because the A1 bit is considered to be immediately to the left

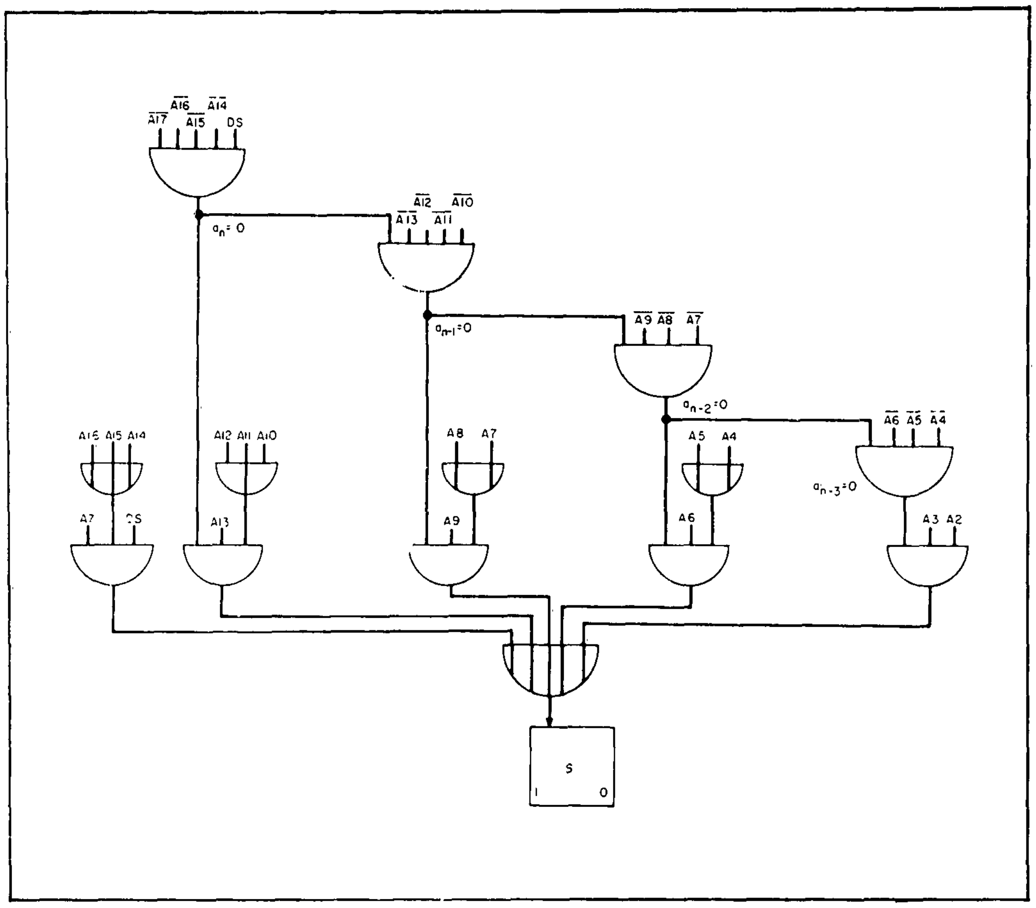


Figure 6-6. Sign Determination

of the A17 bit during this command. The number of shifts is contained in a portion of the instruction register implemented as a countdown counter. When this counter reaches zero, the shifting stops.

The control system requires about 82 flip-flops and a carefully estimated 750 gates in addition to those in the modular adders and multipliers.

6.4 OTHER SYSTEMS STUDIES

6.4.1 Special Memory Considerations

Since a modular multiplication is a one-pulse operation and since this operation can be accomplished in much less time than the cycle time of the fastest core memories, an investigation was made concerning how the problem of inadequate memory speed might be overcome or at least alleviated. Obviously, if faster memories become available, they can be used to particular advantage in a modular machine. Magnetic thin film memories are the most likely candidates, but full scale memories of this type are not likely to be available for several years. Smaller "scratch-pad" or buffer thin-film memories, however, can be obtained now. Such a buffer memory could be used to great advantage in many applications of a modular computer. For example, a comparison of time to multiply two N^2 matrices with and without such a buffer memory has been made. If enough fast buffer storage is available to store a row and a column of the matrices being multiplied, then 2N memory retrievals from the main memory are all the retrievals required to obtain the first element, first row, of the product matrix; N retrievals are needed to obtain the second element, first row, and N more for each of the remaining N-2 elements, first row. Thus $2N + N(N - 1) = N^2 + N$ retrievals are required for the first row and similarly $N^2 + N$ for each additional row. The total retrievals from main memory with buffer storage available is then

$$N(N^2 + N) = N^3 + N^2.$$

Now, if each element of the matrices under multiplication must be retrieved from the main memory each time it is used; i.e., no buffer is available, then each of the $2N^2$ elements of the two matrices must be retrieved N times, requiring $2N^3$ retrievals. Since buffer memory cycle time should be a small fraction of that of the main memory, this time advantage of the buffer memory is significant.

6.4.2 Criteria for Choice of Moduli

The choice of a set of moduli for a modular computer requires that many factors be taken into account. The two fundamental factors are (1) the moduli should be pairwise relatively prime and (2) the product of the moduli must be as large as the required machine range. In addition to these two factors, an examination of our total research effort, i.e., encompassing numerical analysis, systems design, and implementation phases, has revealed eight other significant factors to be considered when choosing a set of moduli for a parallel modular computer. These criteria will also apply in the main for a serial-parallel or serial-by-modulus computer. The short comments given with each of the factors should make each factor clear; in any case, the reader may consult the appropriate paragraph of the report for further details.

a. Maximize memory storage efficiency. Hence each modulus should be just smaller than an integer power of 2 (one modulus can of course be a power of 2). Paragraph 6.2.3.

b. All moduli must be primes if it is required that general linear congruences be solvable. Paragraph 4.2.

c. An integer power of 2 modulus allows a minimal hardware mechanization of its adder and multiplier. Paragraph 7.1.2.

d. If composite moduli; i.e., moduli which are products of primes, are used, one of the factors of one of the composite moduli should be an integer power of 2, and every other factor of the composite moduli should be just less than an integer power of 2. Paragraphs 6.2.3 and 7.1.2.

e. A small number of large moduli gives the required machine range and minimizes mixed radix conversion time. Paragraph 4.1.1 and 6.3.1.

f. A large number of small moduli gives the required machine range and minimizes adder-multiplier hardware. Paragraph 7.

g. 2 or 2 and 5 (or 10) should be moduli or factors of moduli if output conversion to radix 2 or radix 10, respectively, is required. Paragraphs 4.1.4 and 6.3.1.

h. The machine range should be just greater than an integer power of 2 for most rapid convergence in division. Paragraph 4.1.5.

Some of the above factors are obviously mutually exclusive and others are partially redundant. The complete list is given to indicate that all these factors must be considered if the proper choice of moduli for a particular application is to be made. Thus the problem requirements will automatically exclude certain factors from consideration and the remaining ones can be considered in terms of the detailed problem requirements.

6.5 SUMMARY AND RECOMMENDATIONS

6.5.1 Systems and Logic Design Summary

In summary of the systems and logic design studies presented, the modular computer which was designed has the characteristics given in table 6-1. As noted previously, the choice of machine range and system moduli was made for ease of the design studies only. However, the addition of modulus 31 to the given set would give a machine range of about 20 bits, which is adequate for most applications presently under consideration. The addition of modulus 31 would cost approximately

TABLE 6-1

MODULAR ARITHMETIC COMPUTER CHARACTERISTICS

Characteristic	Value
Machine range	±15,015
System moduli	2, 3, 5, 7, 11, 13
Residue coding	LAVR
Mode of operation	Fully parallel
Modular word length	17 bits
Arithmetic unit hardware	425 gates
Control unit hardware	750 gates, 82 FF
Type of control	Synchronous, 2-phase clock
Instruction operation time, clock cycles	Add - 1 Subtract - 1 Multiply - 1 Divide (subroutine) - $\approx$ 100 Determine sign - 10

300 logic gates and 15 flip-flops. Thus a fully parallel modular computer with a range of ±415,415 can be mechanized for about 1600 logic gates and 100 flip-flops using presently available mechanization techniques.

A comparison of the above computer with a conventional fixed point computer should be of value. Table 6-2 gives such a comparison. The modular computer is assumed to have the same clock rate and memory cycle time as the conventional computer. It should be noted that both the clock rate and the memory speed can be significantly improved using present technology.

In fairness to the Airborne Computer, it must be pointed out that this computer has a larger instruction set and was designed to do considerable nonarithmetic data processing, which undoubtedly makes its arithmetic and control hardware somewhat greater. In fairness to the modular computer, it is severely memory-cycle-time limited. It could be organized in a serial-parallel fashion to give the same add and multiply times (3.3 μsec) and twice the divide time (about 300 μsec) with a hardware reduction of 200 to 300 gates.

6.5.2 Recommendations for Further Study

It is recommended that the following areas of systems and logic design be the objects of further studies.

TABLE 6-2

COMPUTER COMPARISON
(Times are in Microseconds)

Characteristic	Westinghouse Airborne Computer	Westinghouse Modular Arithmetic Computer
Machine range	±524, 288	±415, 415
Memory cycle time	3. 3	3. 3
Clock rate	1 MC	1 MC
Add time	6. 0	3. 3
Multiply time	41	3. 3
Divide time	61	≈ 150 (subroutine)
Arithmetic and control gates	1640	1600

6. 5. 2. 1 Modular Memory Addressing

An obvious way to address the core memory in a modular computer is to use the bit positions of one (large) modulus for one address field and similarly for the second address field. However, some of the possible states of the bits are ordinarily nonadmissible; e. g. , 30 and 31 for modulus 29. Since core memories are now constructed in "blocks" which contain an integer-power-of-2 number of words, use of the above - suggested method for modular memory addressing introduces some complications. Preliminary investigations indicate this problem is not a serious one, but it should be investigated in more detail.

6. 5. 2. 2 Classical Simulator

The construction of a modular arithmetic classical simulator would make available a very useful study and design tool. Such a simulator would operate with a modular computer instruction repertoire to solve test problems and thus evaluate the merits of the repertoire, the algorithms used to solve the problems, etc. For example, the exclusion of the magnitude comparison instruction (see paragraph 6. 3. 2) eliminates about 150 control gates. The alternative of programming this instruction could be evaluated on the simulator in terms of running time for test problems. The availability of such a simulator should greatly facilitate the designing of a near-optimal computer configuration for a particular problem, since it would allow various instruction sets and machine organizations to be evaluated in relative detail.

6.5.2.3 Independence and Time Sharing

A modular computer, because of the independence between moduli of modular operations (excluding conversions to and from fixed or mixed radix), has particular properties which are potentially of great value. These properties are involved with the fact that it is possible to (logically or by program) perform mod k logic on mod m equipment provided only that $m \geq k$. If provisions are made for detecting when a mod k unit has failed, then the mod m unit may be time-shared to allow computations of the original accuracy to be performed at a reduced rate. This premise of course requires that equipment be provided to facilitate the detection of failed units and the revision of the program to the slower computing rate.

It is proposed that investigations be conducted in the above area. The properties to be studied are uniquely modular and to our knowledge have not yet been investigated. Favorable results of such studies have obvious applications, particularly in environments where conventional repair is difficult or impossible. Perhaps one can even design a machine which adaptively selects its moduli, based on present computing requirements and on what modular units are functioning at a given time. Certainly the premises of (1) varying machine range with problem requirements and/or (2) isolating failed equipment and using only the remainder are more feasible in a modular arithmetic computer than in any other known machine organization.

7. IMPLEMENTATION STUDIES

7.1 MODULAR ADDER AND MULTIPLIER MECHANIZATIONS

Early in the study, the task was undertaken of investigating and evaluating methods for implementing modular adders and multipliers with core matrices and conventional logic gates. The results of these first investigations indicated that while ferrite core matrices were conceptually simple for mechanizing adders and multipliers, this conceptual simplicity did not take account of hardware considerations. It was found that:

a. A significant amount of hardware is required for decoding binary coded residues modulo m to a (one-of-m wires is hot) code for input to the core matrix, and separate decoding is required for both operands.

b. A logic network is required for encoding the (one-of-m wires is hot) result to a binary coded residue for storing in memory.

c. For a modulus m, 2m core drivers and m sense amplifiers are required for a mod m adder, and similarly for a mod m multiplier. However, if the same matrix is used for both adder and multiplier, drivers and sense amplifiers can be shared.

d. The speed of a core matrix adder or multiplier is limited to approximately the cycle time of a core memory made from the same cores. A reasonable upper limit is a 1-microsecond cycle time as compared to about 1/10 microsecond for semiconductors.

The above disadvantages and limitations of core matrix mechanizations indicated that logic gate mechanizations should be fully and carefully explored, particularly in view of the fundamental speed limitation of cores. Thus, while remaining cognizant of the relation of new modular arithmetic algorithms and general implementation studies to core mechanizations, primary emphasis from the time of these first results to the present has been placed on logic gate mechanization and related minimization techniques.

The approaches to logic gate implementation which have been developed, the values and limitations of each, and promising areas for further work are contained in the following paragraphs.

7.1.1 Direct Implementation

A first approach to the implementation of modular adders and multipliers with logic gates is the straightforward one. For example, to mechanize an adder for modulus m, the summands are assumed available in binary code, truth tables are written to obtain the Boolean functions representing the modular sum, and any of the standard function minimization procedures are applied to reduce the sum functions to minimal form. The same technique is applicable to modular multipliers. This approach has been investigated rather extensively at Westinghouse and elsewhere. It has been determined here that, using this approach, a mod 31 adder may well require several hundred logic gates. The results which indicate this unfortunate circumstance are given below.

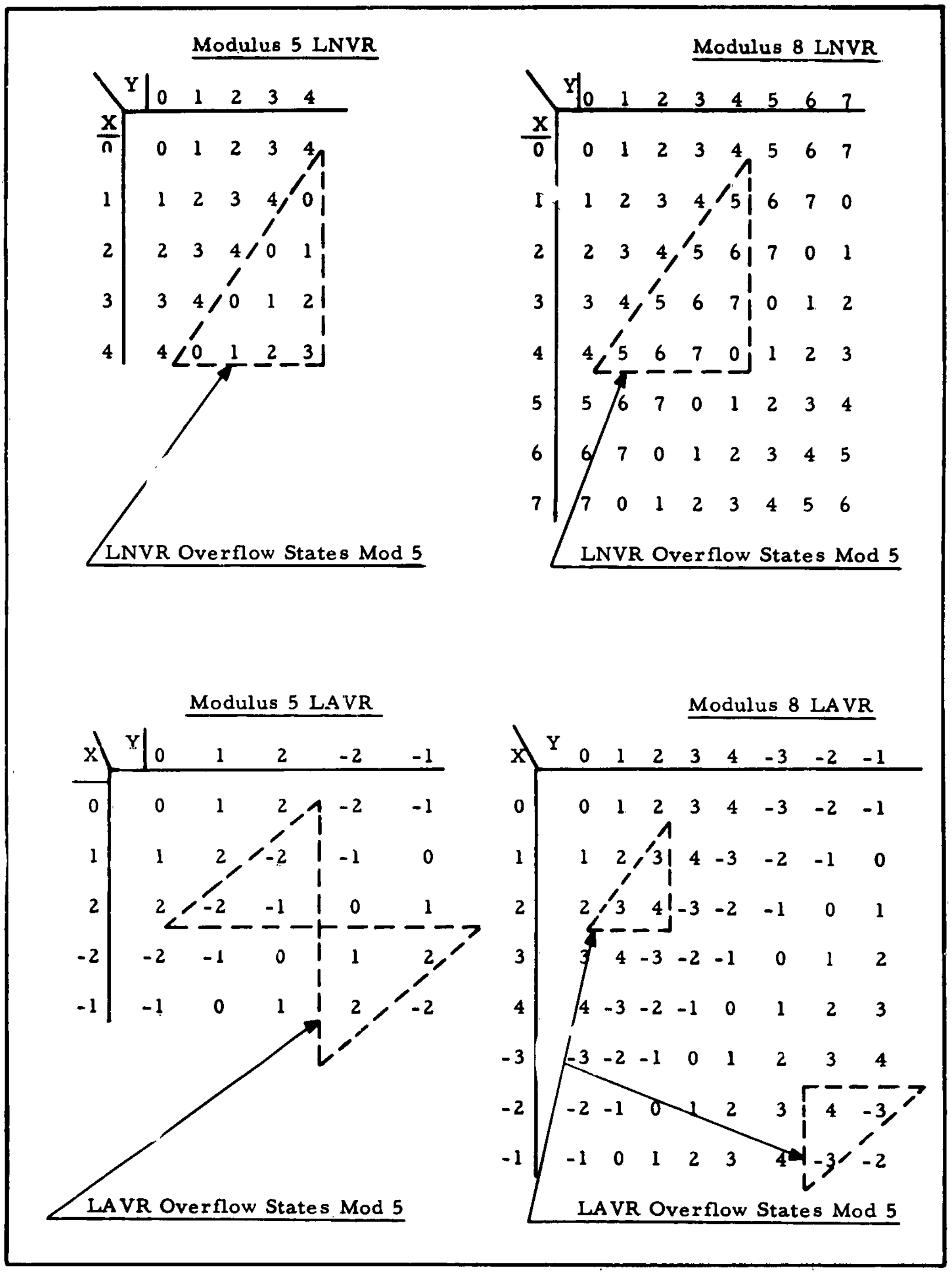
One of the five Boolean functions for the sum mod 31 (specifically that function for the 3rd bit position) was minimized using a SHARE minimization program, PK MIN 4. This function is a 10-variable function containing 505 minterms. After 1 hour of machine time on an IBM 7090, the function had not been reduced to its minimum sum; it had been reduced to a sum of 123 products. Two other programs, PK MIN 2 and LL BAM were tried for equal lengths of time with similar results. If this single function is representative of a practical minimum form for the sum functions modulo 31, then a modulo 31 adder would require something in excess of 600 logic gates for mechanization if two-level logic is used.

The primary reason for the great complexity of the modulo 31 adder is that the five sum functions modulo 31 are each functions of ten variables. The sum functions modulo 32 would also seem to be functions of ten variables. However, it has been shown in reference 1 that a modulo 32 adder can be mechanized for 68 AND-OR and 10 invert gates. The primary reason for this relative simplicity is that instead of five functions each of ten Boolean variables, we have one function of ten, one of eight, one of six, one of four, and finally one of two Boolean variables. In general, if modulus m is an integer power of 2 (and if residues are coded in least nonnegative value residue, LNVR, form), then the Boolean function for bit S_i of the sum will be a function of only the summand input variables A_j and B_j where $j = 0, 1, 2, \ldots, i$. The net result for integer-power-of-2 moduli, 2^n, is that instead of n Boolean sum functions of 2n variables, we have one function S_{n-1} of 2n variables, one function S_{n-2} of 2n-2 variables, and so on, to one function S_o of 2 variables. The analogous result holds for LNVR coded, integer-power-of-2 modulus multipliers. Hence, the mechanization of adders and multipliers for integer-power-of-2 moduli is notably simplified. This property can be directly utilized for only one modulus; i.e., since moduli should be pairwise prime, at most one may be even. If least absolute value residue (LAVR) rather than LNVR binary coding is used, the above results no longer hold in general. However, mechanizations for LAVR integer-power-of-2 moduli are notably simpler than for prime moduli > 2 due to symmetries similar to those of LNVR coding.

Present indications are, then, that the direct implementation approach yields adequately economical mechanizations only for integer-power-of-2 moduli and for relatively small ($M \leq 7$) moduli. It should be pointed out that the computer minimization programs produced reduced functions in 1 minute that were very nearly, and in some cases, exactly the same as the reduced functions obtained 1 hour later. Thus, the function finally produced was a near-minimal sum of products and clearly a practical minimal sum for the given programs. Two other possibilities for obtaining less costly mechanizations are (1) sharing the common parts of the various functions, essentially the multiple output approach, and (2) generating more highly factored functions; e.g., products of sums of products. The first of these approaches was investigated and found to give relatively small reductions in hardware. The second approach might be of significant value if means were available for obtaining these more highly factored functions. However, the general problem of obtaining minimizations of this form is essentially an unsolved one, although some success has been reported recently in reference 2.

7.1.2 Modulus Substitution

The second approach to implementation, modulus substitution, is derived from the observation that modulo k addition can be performed on a modulo m adder, $m > k$, provided means are available for interpreting the result modulo k when "overflow" occurs; i.e., when the absolute value of the sum exceeds $\left|\frac{k-1}{2}\right|$ for LAVR coding and $(k - 1)$ for LNVR coding. Similar remarks hold for multiplication. In figure 7-1 the overflow states of mod 5 addition appear in the outlined triangles. It can be seen that there are 10 overflow states mod 5 for LNVR coding and six overflow states mod 5 for LAVR coding. The number of overflow states is obviously independent of the "parent" modulus, in this case, eight. It is seen that, in general, for modulus k (k odd),



Modulus 5 LNVR

X \ Y	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

Modulus 8 LNVR

X \ Y	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	0
2	2	3	4	5	6	7	0	1
3	3	4	5	6	7	0	1	2
4	4	5	6	7	0	1	2	3
5	5	6	7	0	1	2	3	4
6	6	7	0	1	2	3	4	5
7	7	0	1	2	3	4	5	6

Modulus 5 LAVR

X \ Y	0	1	2	-2	-1
0	0	1	2	-2	-1
1	1	2	-2	-1	0
2	2	-2	-1	0	1
-2	-2	-1	0	1	2
-1	-1	0	1	2	-2

Modulus 8 LAVR

X \ Y	0	1	2	3	4	-3	-2	-1
0	0	1	2	3	4	-3	-2	-1
1	1	2	3	4	-3	-2	-1	0
2	2	3	4	-3	-2	-1	0	1
3	3	4	-3	-2	-1	0	1	2
4	4	-3	-2	-1	0	1	2	3
-3	-3	-2	-1	0	1	2	3	4
-2	-2	-1	0	1	2	3	4	-3
-1	-1	0	1	2	3	4	-3	-2

LNVR Overflow States Mod 5
LNVR Overflow States Mod 5
LAVR Overflow States Mod 5
LAVR Overflow States Mod 5

Figure 7-1. LNVR and LAVR Addition Tables Modulo 5 and 8 - Add X + Y

$$L_n = \text{LNVR overflow states} = \sum_{x=1}^{k-1} x = \frac{k(k-1)}{2};$$

$$L_a = \text{LAVR overflow states} = 2 \sum_{x=1}^{\frac{k-1}{2}} x = \frac{(k-1)(k+1)}{4}.$$

As k becomes large, $L_n \approx 2L_a$, and another advantage of LAVR coding becomes apparent.

The most obvious utilization of the above approach is for allowing serial-by-modulus operation; i.e., by constructing only a parent modular adder and modular multiplier and operating on sufficient smaller moduli, one per machine cycle, to yield the required machine range. For a system with n moduli, an addition or a multiplication would then require n machine cycles. Any convenient combination of fully parallel and serial-by-modulus operation can of course be used. An example of how this technique can be applied to perform mod 5 addition on a mod 8 adder is shown in figure 7-2. The procedure is essentially a two-step one; i.e., perform the addition mod 8, and interpret the sum mod 5. Consequently, a two-phase clock is required, the addition proceeding as follows:

a. Clock Phase One: Input mod 5 summands A and B to mod 8 adder. Sum mod 8 of residues mod 5 is formed and appears in sum register.

b. Clock Phase Two: Interpret the mod 8 result as the sum mod 5. The sum register is reset to the correct sum mod 5, when the sum mod 8 is not already the sum mod 5.

It is seen that 13 logic elements in addition to the mod 8 adder are required to perform the mod 5 addition. For 10 and 18 logic elements, we can perform mod 3 and mod 7 addition, respectively, on a mod 8 adder. This gives a total requirement for mod 3, 5, 7, 8 addition of no more than 80 logic elements. Direct implementation requires about 115 elements for the same four moduli.

A less obvious application of modulus substitution is to combine it with the desirable properties of integer-power-of-2 moduli to obtain adders for moduli of interest. That is, we can directly implement an integer-power-of-2 modular adder and suitably modify it to add mod k for less equipment than that required to implement the mod k adder directly. For example, a mod 13 adder requires about 150 logic gates for direct mechanization and about 102 logic gates for mechanization with a mod 16 adder plus correction logic.

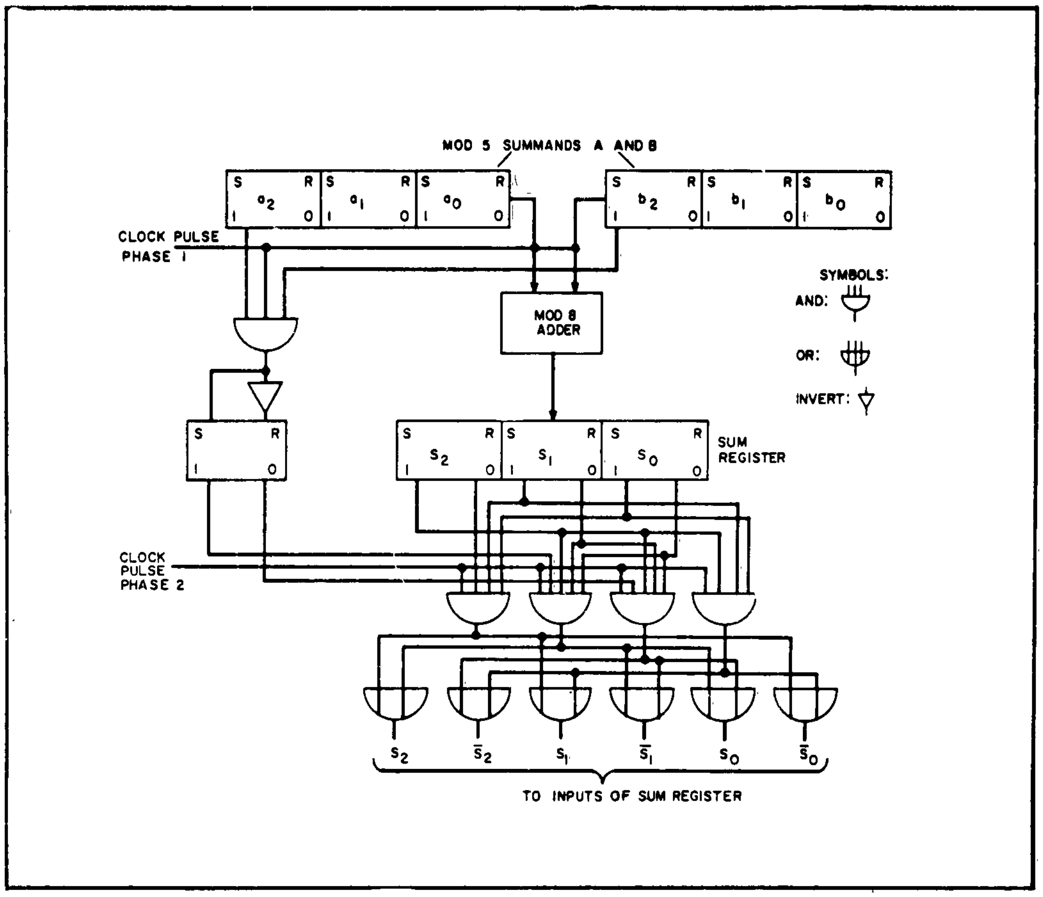


Figure 7-? Mod 8 for Mod 5 Addition

An appreciation for the utilitv of the modulus substitution method for mechanizing modular adders can be obtained from table 7-1. The tabulation is for LAVR mechanizations. For the parallel adder. the adders for moduli 32, 7, 5, and 3 are mechanized directly; those for moduli 11 and 13 are derived from mod 16 adders using modulus substitution. (It should be pointed out that the mod 32 and mod 16 adders are LNVR adders with LAVR inputs logically interpreted as LNVR inputs and vice-versa for the outputs. Thus, these mechanizations can probably be significantly reduced.) The serial-by-modulus adder is derived from a mod 32 adder mechanized directly and used through modulus substitution to add for the moduli 13, 11, 7, 5, and 3, one modulus per clock time. The serial-parallel adder uses a mod 32 direct

TABLE 7-1

DIRECT IMPLEMENTATION AND MODULUS SUBSTITUTION LAVR ADDITION

Modulus	Adder Gates
Parallel Adder Add time = 1 clock time	
32	130
13	102
11	100
7	40
5	26
3	10
$\pi M_i \approx 2^{19}$	Total Gates = 408
Serial by Modulus Adder Add time = 6 clock times	
32	130
13	20
11	18
7	12
5	7
3	4
$\pi M_i \approx 2^{19}$	Total Gates = 191

Modulus	Adder Gates	Modulus	Adder Gates
Serial-Parallel Adder Add time = 3 clock times			
32	130	7	40
13	20	5	13
11	18	3	10
$\pi M_i \approx 2^{19}$ Total Gates = 231			

adder for moduli 32, 13, and 11, and a mod 7 direct adder for moduli 7, 5, and 3. For purposes of comparison with the parallel adder, direct implementation of adders for all these moduli would require about 500 logic gates.

Due to the multiple-overflow problem encountered in modular multiplication, modulus substitution is of limited utility for modular multipliers. A more suitable approach for multipliers is the method to be presented next.

7.1.3 Sign-Magnitude Mechanization

The third approach to modular implementation takes advantage of the symmetry obtained through the use of LAVR coding. Again modulus 5 will be used for an example. As may be seen from the LAVR multiplication table mod 5 in figure 7-3, it is only necessary to mechanize the upper left quadrant of the nonzero portion of the table, giving the signed magnitude of the product. Then, if the signs of the inputs are alike, the signed magnitude is the correct product; otherwise, the sign of the signed magnitude must be complemented. The complete mechanization of a mod 5 multiplier using this approach is shown in figure 7-4. As with the logical substitution method, a two-phase clock is used. The additional logic required to detect and if necessary complement the sign bit of the product consists of 4 AND, 1 OR, 1 INVERT, and 1 set-reset FLIP-FLOP logic elements; total requirement is 16 logic elements. The total requirement for a mod 5 multiplier using direct implementation is 18 logic elements. Thus, the advantage of sign-magnitude mechanization is rather small for the example multiplier. However, because the sign correction logic is identical for all odd moduli, the sign-magnitude method requires no more than about one-half the amount of logic used in direct implementation of multipliers for reasonably large moduli. Precise minimizations have been derived for the sign-magnitude mechanization of multipliers for moduli 7, 11, 13, and 15. The total numbers are 22, 41, 52, and 44 logic elements, respectively. While no precise figures are available for directly implemented multipliers for all of these moduli, our experience indicates that the direct approach would require approximately twice the logic required by the sign-magnitude method.

The sign-magnitude method of implementation can be applied to the mechanization of modular adders, but because the symmetries of an addition table (see figure 7-1) are not as extensive as those of a multiplication table, the economies derived are not as great.

X \ Y	0	1	2	-2	-1
0	0	0	0	0	0
1	0	1	2	-2	-1
2	0	2	-1	1	-2
-2	0	-2	1	-1	2
-1	0	-1	-2	2	1

Figure 7-3. LAVR Multiplication Table Mod 5-Multiply X· Y

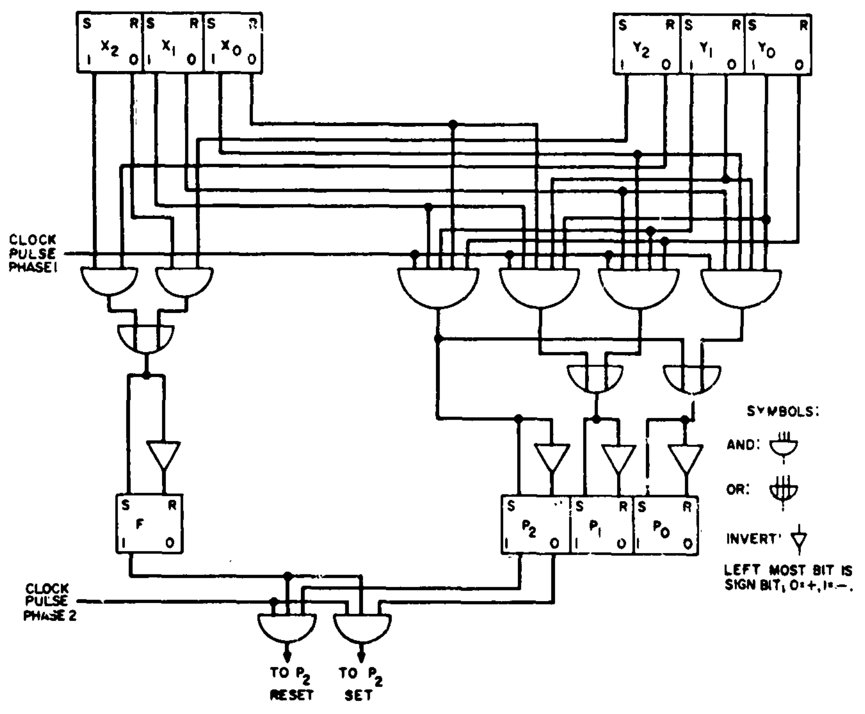


Figure 7-4. Sign-Magnitude Multiplier Mod 5

7.1.4 Range Extension Technique

Using a mod m adder and a mod n adder, mod mn addition may be performed, provided m and n are relatively prime. The same holds for multiplication. At first, this seems of no value, since mn is relatively prime to neither m nor n. However, this approach can be combined with the modulus substitution method to obtain new moduli which are relatively prime to both m and n, thereby extending the range of a modular system.

As an example of this technique, consider that there are available mod 2 and mod 5 adders. Using a truth table-function minimization approach, residues mod 2 and 5 can be converted to a residue mod 10 for 7 AND, 2 OR, and 3 INVERT gates. A mod 10 addition can then be performed by adding residues mod 2 and mod 5 of summands which are each residues mod 10. This implies that the residues mod 10 must be converted to residues mod 2 and mod 5, either in the beginning and stored as such, or when an addition is performed. Using the modulus substitution method, we can use the mod 10 add capability for adding mod 3 and mod 7. The total logic required for this modular adder aggregate (moduli 2, 3, 5, 7) is 48 logic elements. However, including input conversion time, a complete addition requires 3 machine cycles for the addition plus 20 cycles for the determination by conventional means of the residues mod 2, 5 of the residues mod 3 and 7.

To reduce the total add or multiply time using the above approach, the feasibility of using logic networks for directly converting residues from one modulus to another was investigated. Assume now the availability of mod 3 and mod 4 adders and multipliers. We wish to add and multiply for moduli 3, 4, 5, 7, and 11, using the range extension technique to obtain capabilities for the last three moduli. Also, we will use logic networks to provide instantaneous conversion of residues from modulus to modulus as required. Total add or multiply time will be 4 machine cycles. The total logic (direct implementation, LAVR) for the mod 3 and mod 4 adders and multipliers is 22 AND and 6 OR gates. Conversion of mod 3 and mod 4 to mod 12 requires 11 AND, 4 OR, and 4 INVERT gates. (Modulus substitution (for moduli 5, 7 and 11) on first look appears to be simplified over that of paragraph 7.1.3, but such is not the case. For example, to perform mod 11 multiplication on the mod 3, 4 - mod 12 hardware, it is necessary to provide the capability for detecting and correcting for the multiple overflow which can occur in multiplication. Preliminary investigations have not yielded economical mechanizations for correcting multiple overflow, but this is not to say such mechanizations cannot be developed.)

The logic required for input conversion of mods 5, 7, and 11 to residues mod 3 and 4 is 21 AND, 10 OR, and 10 INVERT gates. Now, if the residue mod 5 (or 7, or 11) is stored as a residue mod 5 (or 7, or 11) then an addition or a multiplication will often require that two determinations of the residues mod 3 and 4 of the residue mod 5 (or 7, or 11) be made. If the above logic network (21 AND, 10 OR, 10 INVERT GATES) performs this conversion and a total add or multiply time of 4 machine cycles is required, two such logic networks will be needed. Figure 7-5 shows a likely configuration for the adder-multiplier just discussed.

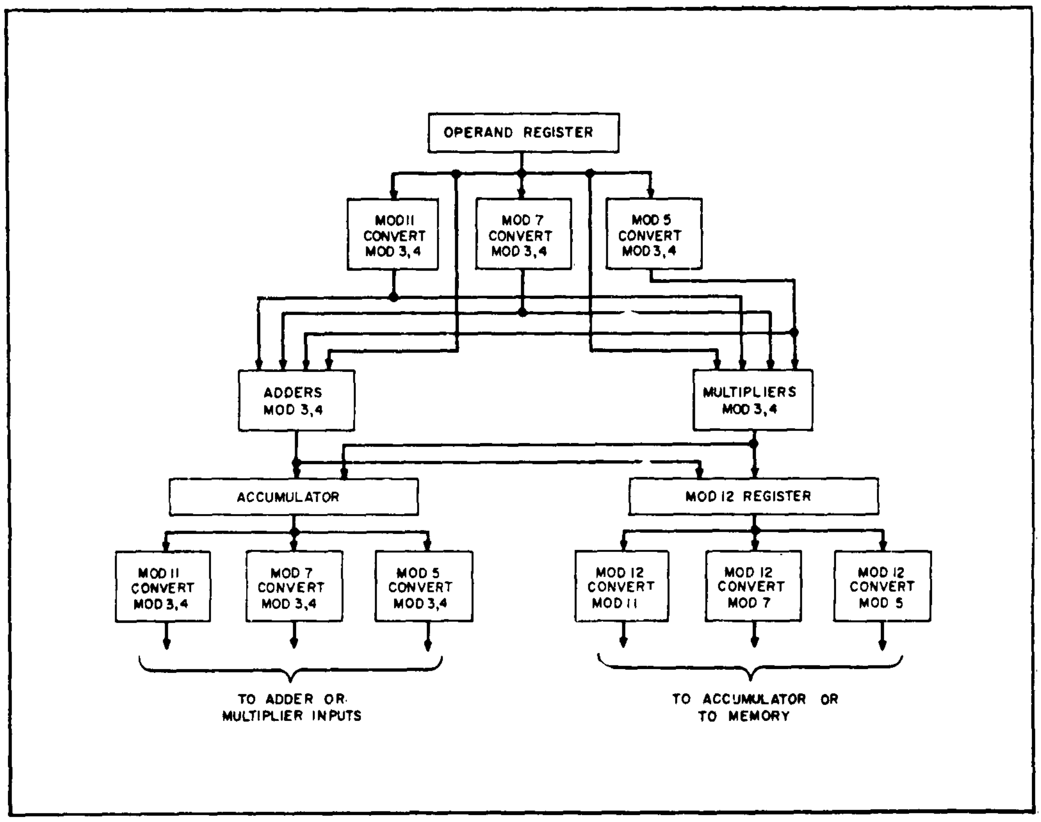


Figure 7-5. Mod 3, 4 and 5, 7, 11 Adder-Multiplier

The hardware requirement for the above adder-multiplier excluding mod substitution logic is 129 logic elements. If the residues mod 5, 7, and 11 are stored as residues mod 3 and mod 4 of residues mod 5, 7, and 11, this requirement is cut by 41 logic elements, at the expense of two bits added to the modular word length.

The factor which makes the above technique significant is that much of the logic is used for both addition and multiplication. If logic techniques to economically substitute one modulus for another in multiplication can be developed, range extension will provide more economical mechanizations than any now available, particularly for large moduli (> 32).

A variation on the range extension technique is to utilize pairs of moduli as pseudosingle moduli for reducing the time required for sign determination. For example, given the moduli 2, 3, 5, 7, 11, and 13, the arithmetic unit can be designed to appear to be operating with moduli 26, 33, and 35 during mixed radix conversion. Sign determination time is reduced from 10 to 4 machine cycles, or from 5 to 2 cycles depending on how mixed-radix conversion is obtained.

7.1.5 Programmed Arithmetic

A method will be presented in which existing hardware is used to extend the machine range through programming subroutines; i.e., a "microprogram" is used to program a mod m unit, $m > k$, to do mod k arithmetic with no additional circuitry. In the following, $\odot$ and $\oplus$ will be used to designate mod k operations. Least absolute value representation is assumed.

The case $k = 2$, $m = 3$ is quite simple.

$$x \odot y = xy$$

$$x \oplus y = x + y + xy = x + y - 2xy$$

When $m = 5$,

$$k = 2 \quad \begin{array}{l} x \odot y = xy \\ x \oplus y = x + y - 2xy \end{array}$$

$$k = 3 \quad \begin{array}{l} x \odot y = xy \\ x \oplus y = x + y + xy\,(x + y) \end{array}$$

When $m = 7$,

$$k = 2 \quad \begin{array}{l} x \odot y = xy \\ x \oplus y = x + y - 2xy \end{array}$$

$$k = 3 \quad \begin{array}{l} x \odot y = xy \\ x \oplus y = x + y + 2xy\,(x + y) \end{array}$$

$$k = 4 \quad \begin{aligned} x \odot y &= xy + 2xy(x-1)(y-1)(xy-1) \\ x \oplus y &= x + y + 3xy(x+y)(x+y-1)(x+y-2)\left[(x+y)^2+1\right] \end{aligned}$$

$$k = 5 \quad \begin{aligned} x \odot y &= xy + 2xy(x^2-1)(y^2-1) \\ x \oplus y &= x + y + 2(x+y)\left[(x+y)^2+3\right]\left[(x+y)^2-1\right](3-xy) \end{aligned}$$

For general m,

$$k = 2 \quad \begin{aligned} x \odot y &= xy \\ x \oplus y &= x + y - 2xy \end{aligned}$$

$$K = 3 \quad \begin{aligned} x \odot y &= xy \\ x \oplus y &= x + y + axy(x+y) \end{aligned}$$

where

$2a = -3 \bmod m.$

Apparently multiplication is simpler than addition, but for $k > 3$ the programs appear to be too long. On the other hand, no additional hardware (other than storage) is required for these programs. A single mod m unit can be used to increase the machine range by a factor of $(2) \cdot (3) = 6$ at the cost of tripling MULTIPLY time and multiplying ADD time by 10 if 2 and 3 were not used as original moduli. On the other hand, if mod m_1 and m_2 units are used to compute mods 2 and 3 simultaneously, then this increase in machine range is available at a cost of doubling MULTIPLY time and multiplying ADD time by a factor of 5. (If m = 5 is used for mod 3 arithmetic, this latter factor is reduced to 4.)

7.2 MOLECULAR LOGIC ELEMENTS

A study has been made of molecular logic elements for the purpose of determining availability, applicability, and reliability of such elements as might be used in the mechanization of a modular computer. Several types of currently available single-element logic gates have been investigated and each has been found to be applicable to some degree. Reliability data are given here for one type, Fairchild Micrologic elements. Life-test data of Micrologic elements validate a failure rate of better than 0.01 percent per 1000 hours, to a 60 percent confidence limit. This reliability figure is sufficiently high to yield for a medium size computer employing 2000 such elements, for example, the following approximate reliability.

Mean time between failures:

$$MTBF = \frac{1}{(\text{parts})(\text{failure rates})} = \frac{1000 \text{ hours}}{2(10^3)\ 10^{-4}}$$

$$= 5000 \text{ hours.}$$

Operating at 90 percent reliability:

R: Reliability

λ: System failure rate = $\sum$ part failure rates

t: time period involved

$$R = e^{-\lambda t}$$

$$-\text{Ln } 0.90 = \frac{-2(10^{-1})\ t}{1000 \text{ hours}}$$

$$t \approx 500 \text{ hours.}$$

The above reliability figures are order of magnitude estimates and are given for general information only. The failure rate assumed for the Micrologic elements is higher than the actual one and was obtained by assuming one failure had occurred at the end of the life test period, whereas actually none had occurred.

Other molecular logic elements which were investigated did not have available life test data, but this data in some cases will be available soon. Since two of the manufacturers concerned, Westinghouse and Texas Instruments, are presently developing molecular elements for the Minuteman program, the adequate reliability of such elements seems further assured.

7.3 SUMMARY AND RECOMMENDATIONS

The methods which have been presented for mechanizing modular adders and multipliers demonstrate that modular arithmetic computers can be economically implemented with logic gates. These methods also indicate that speed and complexity may be exchanged in an almost continuous manner, from a programmed serial-by-modulus machine to a fully parallel machine of extreme speed. Finally, these studies of implementation methods have given major reductions in the hardware required for mechanizations, and it is felt that further reductions will result from continued studies.

In summary of our implementation studies, then, we will present first a review of the best approaches now available and second a discussion of directions for further work.

7.3.1 Present Status of Implementation Techniques

If a modular adder for modulus m is to be mechanized, we have at our disposal two "best" techniques. If m is less than 8 or any integer power of 2, then the mod m adder can be mechanized most economically through direct implementation as discussed in paragraph 7.1.2. If m does not meet the above conditions, then the modulus substitution method of paragraphs 7.1.3 and 7.1.5 is most economical.

If a modular multiplier is to be mechanized, the sign-magnitude technique of paragraph 7.1.4 requires minimal hardware. It can be noted that judicious choice of moduli, for example, modulus 31, allows the signed magnitude mechanization to be very nearly the same as a LNVR mechanization for modulus 16 which, of course, is notably simple.

7.3.2 Directions for Further Study

7.3.2.1 Minimization of Boolean Functions

The use of existing computer programs to obtain minimizations of Boolean functions has proven unsatisfactory in modular arithmetic implementation studies. While this fact is in part due to the large number of variables present, more significant is the observation that the class of Boolean functions of n variables generated by modular adders and multipliers is most likely a very small subset of the complete set of 2^{2^n} functions of n variables. Since the computer programs used were all written for the complete set of functions, it is reasonable to assume that the modular function minimizations obtained suffered from the severely overgeneral nature of the programs. It is recommended that a study be made, first to determine the subclass of functions to which modular functions belong, and second, to exploit the special properties of this subclass, if possible, to obtain more economical mechanizations through both conventional minimization techniques and new computer programs.

7.3.2.2 Multiphase and Multicycle Operations

The method of sign-magnitude mechanization gives a major hardware reduction by trading a second phase of a clock cycle for the hardware it replaces. The serial-by-modulus approach made possible by modulus substitution also trades time for hardware. Further methods of this type should be investigated. For example, a multiple-sign coding offers some promise of further hardware reductions for modular multipliers.

7.3.2.3 Range Extension and The Sharing of Logic Between Adders and Multipliers

The technique of range extension (i.e., utilizing pairs of small moduli to obtain large moduli and thereby extending the range) has most interesting properties. Consider, for example, the moduli 7 and 8 extended to modulus 55. We then have mod 7, 8 adders and multipliers, a mod 7, 8 to mod 56 converter, a mod 56 to mod 55 correcting converter (for add and multiply), and a mod 55 to mod 7, 8 converter. If a mod 56 adder and multiplier had been mechanized directly, a total of 12, 12-variable Boolean functions would have been implemented. With the range extension technique, we mechanized instead 12, 6-variable functions for the mod 7, 8 adders and multipliers, 10, 6-variable functions for the 7, 8 $\rightarrow$ 56 and 55 $\rightarrow$ 7, 8 converters, plus an as yet unspecified logic network for the mod 56 to mod 55 correcting converter.

The network to correct the mod 56 sum to the mod 55 sum is simply a modulus substitution network as described in paragraph 7.1.3. The network to correct the mod 56 product to the mod 55 product, however, is not so simple. Basically, this network must have the capability to determine if the product of the mod 55 inputs has overflowed mod 56 and if so how much. The mod 55 inputs must be examined for this information. An economical method for accomplishing this detection and correction should be developed.

If the correcting converter just described is available, then range extension becomes an exceedingly powerful technique. For example, all hardware used in converting (7, 8 $\rightarrow$ 56, 56 $\rightarrow$ 55, and 55 $\rightarrow$ 7, 8) is used for both adding and multiplying. Further, it then becomes very easy for the program to specify the range of the machine. That is, a correcting converter should basically be capable of correcting mod 56 to other moduli, say 53, 51, and 47. Thus, single-precision computation can proceed very rapidly, while very high precision is available at lower speed.

7.3.2.4 Isomorphic Relations

Considering the operation of addition alone, the residues mod m form a cyclic group. The direct sum of 2 such groups for moduli m_1 and m_2; that is, the set of pairs (x_1, x_2) with addition performed mod m_1 for the first component and mod m_2 for the second component, is a cyclic group of order $m_1 m_2$ isomorphic to the group of residues mod $m_1 m_2$ under the operation of addition. That is, the set of pairs simply acts as a code for the residues mod $m_1 m_2$ and conversely, so that if

$X \longleftrightarrow (x_1, x_2)$

$Y \longleftrightarrow (y_1, y_2)$

then

$XY \longleftrightarrow (x_1 + y_1, x_2 + y_2)$.

In fact, this is the basic idea of mod arithmetic.

It is well known that a cyclic group contains cyclic subgroups of all orders which divide the order of the group.

On the other hand, the set of residues mod p excluding 0 forms a cyclic group of order p - 1 under the operation of multiplication mod p. Now any two cyclic groups of the same order are isomorphic. Hence, if p is a prime such that p - 1 divides $m_1 m_2$, then the multiplicative group mod p is a subgroup of the additive group mod $m_1 m_2$ and hence of the direct sum of the additive groups mod m_1 and m_2. For example, with $m_1 = 8$, $m_2 = 15$, $p = 41$, the mapping shown below is an isomorphism.

Mod 41 Residue	Mod (8, 15) Code	Mod 41 Residue	Mod (8, 15) Code
1	(0, 0)	21	(2, 12)
2	(6, 3)	22	(7, 12)
3	(5, 0)	23	(4, 3)
4	(4, 6)	24	(7, 9)
5	(2, 6)	25	(4, 12)
6	(3, 3)	26	(3, 6)
7	(5, 12)	27	(7, 0)
8	(2, 9)	28	(1, 3)
9	(2, 0)	29	(5, 6)
10	(0, 9)	30	(5, 9)
11	(1, 9)	31	(4, 9)
12	(1, 6)	32	(6, 0)
13	(5, 3)	33	(6, 9)
14	(3, 0)	34	(1, 12)
15	(7, 6)	35	(7, 3)
16	(0, 12)	36	(6, 6)
17	(3, 9)	37	(0, 6)
18	(0, 3)	38	(1, 0)
19	(3, 12)	39	(2, 3)
20	(6, 12)	40	(4, 0)

If for example, one wishes to multiply (18) (37) mod 41, one adds their mod (8, 15) counterparts

(0, 3) + (0, 6) = (0, 9) mod (8, 15)

and finds that 10 is the counterpart of (0, 9). (Actually, (18) (37) = 16(41) + $\underline{10}$.)

Thus, except for multiplication by 0, which is trivial, the entire multiplication table mod 41 is contained in the mod (8, 15) addition tables. In fact, subgroups of the (8, 15) additive group are isomorphic to the multiplicative groups mod 3, 5, 7, 11, 13, 31, 41, and 61, at least.

The significance of this is that mod 41 multiplication is a priori considered to be difficult to implement, while mod 8 and 15 adders are simple to mechanize, and yet the latter pair of addition tables contains the mod 41 multiplication table, as well as others. It is therefore recommended that mechanization studies attempt to capitalize on this algebraic structure. Note that such studies are purely modular in nature since such structure is lacking in conventional (i.e., nonmodular) coding. Two directions of effort are indicated:

a. The use of the isomorphism directly to allow the use of the adders or parts thereof together with coders and decoders or table look-up, to perform multiplication.

b. The use of the isomorphic relations in minimizing the Boolean functions for the multiplication tables with specific implementation of the multiplication function.

7.3.2.5 Psuedo-Single Moduli

The technique of using pairs of moduli as psuedo-single moduli during mixed radix conversion was briefly discussed in paragraph 7.1.5. The example given there used moduli 2, 3, 5, 7, 11, and 13 which were paired as (2) · (13) = 26, (3) · (11) = 33, and (5) · (7) = 35. This example will be further considered here. Let the residues mod 26 be stored as residues mod 2 and mod 13 of residues mod 26, and, similarly, for mod 33 as mod 3, 11 and mod 35 as mod 5, 7. In the mixed-radix conversion, the d_{kj}'s must be available in special form; e.g., if modulus 26 is to be eliminated first, we must perform the operations.

$$(X_{33} - X_{26}) \cdot d_{26,\ 33} \text{ and } (X_{35} - X_{26}) \cdot d_{26,\ 35}.$$

Since the operation $(X_{33} - X_{26}) \cdot d_{26,\ 33}$, for example, is actually performed modulo (3, 11), $d_{26,\ 33}$ must be used in the form of $d_{26,\ 33}$ (mod 3) and $d_{26,\ 33}$ (mod 11). Also, since residues mod 2, 13 will have to be converted to residues mod 3 and mod 11, extra hardware will be required for this conversion. In addition to the mod 2, 3, 5, 7, 11, and 13 adders and multipliers, the following hardware must be provided:

one mod 2, 13 to mod 3, 11 converter

one mod 2, 13 to mod 5, 7 converter

one mod 3, 11 to mod 5, 7 converter

The mechanization of these converters is straightforward and requires that a total of 12, 5-variable and 6, 6-variable functions be mechanized.

Several questions remain unanswered concerning the above technique:

a. What is the optimum grouping of the pairs of moduli for minimal hardware realization of the technique?

b. Can more than two moduli be grouped economically?

c. Is this technique, in general, more economical for reducing mixed radix conversion time than wire-twisting, or than employing a smaller number of single large moduli?

Obviously, the above questions should be investigated, since they are directly concerned with reducing the operation time of such a fundamental modular operation.

The high degree of interdependence between all of the above-proposed investigations and particularly between 7. 3. 2. 1 and 7. 3. 2. 4 will be recognized.

8. REFERENCES

1. Garner, H. L., et al., "Residue Number Systems for Computers," ASD Technical Report 61-483, University of Michigan, October 1961.

2. Driese, Edward C., George E. Glen, and Ralph E. Young, Jr., "Computer Applications of Residue Class Notations," ASD Technical Report 61-189, September 1961.

3. Shapiro, H. S., "Some Remarks on Modular Arithmetic and Parallel Computation," Mathematics of Computation, 16, (April 1962), 218-222.

4. Trench, W. F., "On Periodicities of Certain Sequences of Residues," American Mathematical Monthly, LXVII, 7, (1960).

5. Frazer, R. A., W. J. Duncan, and A. R. Collar, "Elementary Matrices," Cambridge, (1950), 37-39.

6. Fadeeva, V. N., "Computational Methods of Linear Algebra," Dover Publications, Inc., New York, (1959), 167.

7. Vinogradov, I. M., "Elements of Number Theory," Dover Publications, Inc., 1954.

8. Cheney, Philip W., "A Digital Correlator Based on the Residue Number System," IRE Trans. on Electronic Computers, EC-10, 1, (March 1961).

9. Lawler, Eugene L., "Minimal Boolean Expressions with More than Two Levels of Sums and Products," Switching Circuit Theory and Logical Design, Proceedings of the Third Annual Symposium, Chicago, Ill., October 7-11, 1962, Published by the AIEE, (1962), 49-59.

10. Pei, M. L., "A Test Matrix for Inversion Procedures," Communications of the Association for Computing Machinery, 5, 10, (1962).

11. "Modular Arithmetic Computing Techniques," Interim Engineering Report Number 2, Westinghouse Electric Corporation, 25 August 1962.

APPENDIX

PROOF OF CONVERGENCE OF SQUARE-ROOT PROCEDURE (See Paragraph 4.1.6)

The proof of convergence is by induction and is divided into two cases: In Case I, p is odd, $x_1 = 2^{(p-1)/2}$, and $k_i = 2^{z_i - (p+3)/2}$; while in Case II, p is even, $x_1 = 2^{p/2}$, and $k_i = 2^{z_i - (p+2)/2}$. The basic idea of the proof is to show that z_i decreases as i increases. To show this, it is necessary to prove that all x_i lie between certain bounds.

For instance, in Case I, we must show that $2^{(p-1)/2} = x_1 \leq x_i < 2^{(p+1)/2}$. We know that x_1 is less than a, so $x_2 > x_1$ and, so long as $x_i < a$, $x_{i+1} > x_i$. Hence, the only way for x_{i+1} to be less than x_1 is that x_i become greater than a for some i. What we intend to show is that, when such a "reversal of direction" occurs in the sequence, $x_1, x_2, \ldots,$ then z_i is decreased sufficiently to guarantee that all succeeding x_i will be greater than x_1. That all x_i are less than $2^{(p+2)/2}$ is shown directly. We now proceed with the proof of Case I.

Case I

Our induction hypothesis is that $z_i \leq z_{i-1} \leq p - 1$ and $2^{(p-1)/2} \leq x_i < 2^{(p+1)/2}$. We have already shown in paragraph 4.1.6 that these conditions are satisfied for $i = 1$ (assuming that $z_0 = p - 1$). We now assume they are satisfied for an arbitrary $i \geq 1$ and prove this implies they are satisfied for $i + 1$.

First, we obtain bounds on $|a - x_i|$. From the definition of z_i and the induction hypothesis, we have $2^{z_i} \geq |a^2 - x_i^2| = (a + x_i)\,|a - x_i| > (2 \cdot 2^{(p-1)/2})\,|a - x_i| = 2^{(p+1)/2}\,|a - x_i|$, and $2^{z_i - 1} < |a^2 - x_i^2| = (a + x_i)\,|a - x_i| < (2^{p/2} + 2^{(p+1)/2})\,|a - x_i| < 2^{(p+3)/2}\,|a - x_i|$. Hence, $2^{z_i - (p+5)/2} < |a - x_i| < 2^{z_i - (p+1)/2}$.

Next, we show that $x_{i+1} < 2^{(p+1)/2}$. If $x_i > a$, then $x_{i+1} = x_i - 2^{z_i - (p+3)/2} < x_i < 2^{(p+1)/2}$, by the induction hypothesis. On the other hand, if $x_i < a \le 2^{p/2}$, then $x_{i+1} = x_i + 2^{z_i - (p+3)/2} < 2^{p/2} + 2^{p - 1 - (p+3)/2}$ $= 2^{p/2} + 2^{(p-5)/2} < 2^{(p+1)/2}$.

Now, if $(a - x_i)$ and $(a - x_{i+1})$ have opposite signs - that is, if the sequence "reverses direction" at x_i -, then it follows that $2^{z_i - (p+3)/2} > |a - x_i|$ and $|a - x_{i+1}|$ $= 2^{z_i - (p+3)/2} - |a - x_i| < 2^{z_i - (p+3)/2} - 2^{z_i - (p+5)/2} = 2^{z_i - (p+5)/2}$.

Therefore, $|a^2 - x_{i+1}^2| = (a + x_{i+1})\,|a - x_{i+1}|$ $< (2^{p/2} + 2^{(p+1)/2})\, 2^{z_i - (p+5)/2} < 2^{(p+3)/2} \cdot 2^{z_i - (p+5)/2} = 2^{z_i - 1}$, and $z_{i+1} \le z_i - 1$.

Moreover, if $(a - x_i)$ and $(a - x_{i+1})$ have the same signs, then $|a - x_{i+1}|$ $= |a - x_i| - 2^{z_i - (p+3)/2} < 2^{z_i - (p+1)/2} - 2^{z_i - (p+3)/2} = 2^{z_i - (p+3)/2}$ and $|a^2 - x_{i+1}^2| = (a + x_{i+1})\,|a - x_{i+1}| < (2^{p/2} + 2^{(p+1)/2}) \cdot 2^{z_i - (p+3)/2}$ $< 2^{(p+3)/2} \cdot 2^{z_i - (p+3)/2} = 2^{z_i}$. Hence, $z_{i+1} \le z_i$, and if $z_{i+1} = z_i$, it follows from the bounds on $|a - x_i|$ that the sequence reverses direction at x_{i+2}; that is, $(a - x_{i+1})$ and $(a - x_{i+2})$ have opposite signs. Therefore, $z_{i+2} \le z_i - 1$.

We have now shown that z_i decreases as i increases, which implies that x_i approaches a as i increases. It remains to be shown that $x_i \ge x_1 = 2^{(p-1)/2}$, for all $i > 1$. We know that $x_1 < a$, so if the sequence does not stop at x_1, then x_2 must be greater than x_1. Obviously, then there is no danger of x_i becoming less than x_1 unless the sequence reverses direction for some i. But when that happens, z_i decreases by at least 1 as we have shown above. Hence, if x_i is the first element of the sequence that is greater than a, then $z_i \le z_{i-1} - 1$, which implies that $x_{i+1} > x_1$. Now if $z_i = z_{i+1} = z_1 - 1$, then $x_{i+2} \ge x_1$, and another reversal of direction occurs at x_{i+2}. And if either of z_i or z_{i+1} is less than $z_1 - 1$, then it follows from the above arguments that all of the elements, $x_{i+1}, x_{i+2}, \ldots$, are strictly greater than x_1. Either way, we see that, because z_i must decrease by at least 1 for every two successive terms of the sequence, all x_i's are $\ge x_1$. This completes the proof that z_{i+1} and x_{i+1} satisfy the induction hypothesis; hence, by the induction principle, all x_i's and z_i's satisfy the induction hypothesis for $i \ge 1$.

In Case II, the basic idea of the proof is the same as in Case I, but the situation is complicated by the fact that the "bounds" in x_i are reversed; that is, it is relatively easy to show directly that $x_i > 2^{(p-2)/2}$ for all $i \ge 1$, but an almost "circular" argument must be used to show that $x_i < x_1 = 2^{p/2}$. Hence, we first consider the situations in which $x_i > a$, since this assures us that x_{i+1} is bounded (by x_i). Since $x_1 \geq a$, we can get enough results about the behavior of the sequence x_1, x_2, . . . , in these situations to complete the proof for the remaining possible situations; i.e., these in which $x_i < a$. We now proceed with the proof.

Case II

Our induction hypothesis is that $z_i \leq z_{i-1} \leq p - 1$ and $2^{(p-2)/2} < x_i \leq 2^{p/2}$, which we have already shown to be true for i = 1. As in Case I, we show that if we assume the hypnothesis is true for some arbitrary i, then it is also true for i + 1.

We first obtain bounds on $|a - x_i|$. We have $2^{z_i} \geq |a^2 - x_i^2| = (a + x_i)\,|a - x_i| > (2^{(p-1)/2} + 2^{(p-2)/2})\,|a - x_i| > 2^{(p/2)} \cdot |a - x_i|$, and $2^{z_i - 1} < |a^2 - x_i^2| = (a + x_i) \cdot |a - x_i| < (2 \cdot 2^{p/2}) \cdot |a - x_i| = 2^{(p+2)/2} \cdot |a - x_i|$. Hence,
$2^{z_i - (p+4)/2} < |a - x_i| < 2^{z_i - (p/2)}$.

Next, we consider the case where $(a - x_i)$ and $(a - x_{i+1})$ have the same sign. Let us assume only that $x_{i+1} < x_i$. Then $|a - x_{i+1}| = |a - x_i| - 2^{z_i - (p+2)/2} < 2^{z_i - (p/2)} - 2^{z_i - (p+2)/2} = 2^{z_i - (p+2)/2}$, and from our assumptions it follows that $(a + x_{i+1}) \leq 2x_{i+1} < 2x_i \leq 2^{(p+2)/2}$. Hence, $|a^2 - x_{i+1}^2| = (a + x_{i+1})\,|a - x_{i+1}| < 2^{(p+2)/2} \cdot 2^{z_i - (p+2)/2} = 2^{z_i}$. Therefore, in this subcase, $z_{i+1} \leq z_i$.

Now we consider an analogous situation for the case where $(a - x_i)$ and $(a - x_{i+1})$ have opposite signs; that is, we assume only that $x_i > a$. Then $2^{z_i - (p+2)/2} > |a - x_i|$ and $|a - x_{i+1}| = 2^{z_i - (p+2)/2} - |a - x_i| < 2^{z_i - (p+2)/2} - 2^{z_i - (p+4)/2} = 2^{z_i - (p+4)/2}$. Also, $(a + x_{i+1}) < 2x_i \leq 2^{(p+2)/2}$, from which we have $|a^2 - x_{i+1}^2| = (a + x_{i+1})\,|a - x_{i+1}| < 2^{(p+2)/2} \cdot 2^{z_i - (p+4)/2} = 2^{z_i - 1}$.

Hence $z_{i+1} \leq z_i - 1$.

But since x_1 must be greater than or equal to a, all succeeding x_i, if any, must be less than x_1 unless the sequence "reverses direction." If this happens, z_i decreases by at least 1, so we can apply an argument similar to that used in Case I. Suppose the sequence reverses direction from decreasing to increasing at x_i; that is, $a > x_i$ and $a < x_{i-1}$. Then, $z_i \lesssim z_{i-1} - 1$, as we have shown above, and therefore $x_{i+1} < x_{i-1}$. Also, if $z_i = z_{i+1} = z_{i-1} - 1$, then $x_{i+2} \leq x_{i-1}$; and if either z_i or z_{i+1} are less than $z_{i-1} - 1$, then all succeeding elements of the sequence are $< x_{i-1}$. Hence, the above arguments for the situation where $x_i > a$ are also valid for the situation where $x_i < a$, since $x_i, x_{i+1}, x_{i+2}, \ldots$, are $\leq x_{i-1}$ for the "point of reversal" x_i, which is $\leq 2^{p/2}$ by the induction hypothesis. Hence, in general, $z_{i+1} \leq z_i$ whenever $(a - x_i)$ and $(a - x_{i+1})$ have the same sign; and $z_{i+1} \leq z_{i-1}$ whenever $(a - x_i)$ and $(a - x_{i+1})$ have opposite signs.

Now all that remains is to show that $x_{i+1} > 2^{(p-2)/2}$. To do this, we first show that, if $z_1 = p - 1$ and $x_2 > a$, then $z_2 < p - 1$. For $(a + x_2) < 2\,x_2 < 2\,x_1 = 2^{(p+2)/2}$ and $|a - x_2| = x_2 - a < 2^{p/2} - 2^{(p-4)/2} - 2^{(p-1)/2} < 2^{(p-8)/2}$; hence, $|a^2 - x_2^2| = (a + x_2)\,|a - x_2| < 2^{(p+2)/2} \cdot 2^{(p-8)/2} = 2^{p-3}$, from which we have $z_2 < p - 2$. Therefore, for $i \geq 2$, $z_i \leq p - 2$. Now if $x_i < x_{i-1}$, x_{i-1} must be $> a$, which implies that $x_{i-1} > 2^{(p-1)/2}$. Hence,

$$x_i = x_{i-1} - 2^{z_i - (p+2)/2} > 2^{(p-1)/2} - 2^{(p-6)/2} > 2^{(p-2)/2}.$$

By the induction hypothesis and the results proven above, this completes the proof.

What we have proven in both cases above is that the sequence $x_1, x_2, \ldots$, converges to a with an increase in accuracy of at least one binary bit per two terms of the sequence. In actual practice, the convergence is considerably faster than this, since the theoretical "worst cases," used in the above proofs, never occur.

Aeronautical Systems Division, Dir/Avionics, Electronic Technology Laboratory, Wright-Patterson Air Force Base, Ohio.
Rpt Nr ASD-TDR-63-280. MODULAR ARITHMETIC COMPUTING TECHNIQUES. Summary Technical Report, May 63, 122p. incl illus., tables, 11 refs.

Unclassified Report

Modular arithmetic concepts and associated computation techniques and organization are outlined. Fundamental operations of modular arithmetic discussed include sign or relative magnitude determination and division, mathematical solution using modular arithmetic, techniques for efficient mechanization of modular arithmetic adders and multipliers, and organization and control of a modular arithmetic computer.

(over)

Numerical analysis studies yielded novel results including the introduction of signed residues, overflow detection techniques, and a division algorithm 3 times as fast as any previously disclosed. A square-root algorithm which is considerably faster than the Newton-Raphson algorithm is discussed.

Implementation techniques are described for trading speed for complexity, programming a computer to extend its range, and design techniques for reducing component counts in adders and multipliers.

A functional simulation of modular arithmetic computation in a conventional computer is described, and statistical data on the operation of the various algorithms is included.

1. Modular arithmetic
2. Residue number system
3. Digital computers
4. Computer logic
5. Mathematics

I. AFSC Project 7062 Task 706205
II. Contract No. AF33(657) 7899
III. Westinghouse Electric Corporation, Air Arm Division, Baltimore, Md.
IV. Contractor's Report No. 1274A
V. Not aval fr OTS
VI. In Astia Collection

Aeronautical Systems Division, Dir/Avionics, Electronic Technology Laboratory, Wright-Patterson Air Force Base, Ohio.
Rpt Nr ASD-TDR-63-280. MODULAR ARITHMETIC COMPUTING TECHNIQUES. Summary Technical Report, May 63, 122p. incl illus., tables, 11 refs.

Unclassified Report

Modular arithmetic concepts and associated computation techniques and organization are outlined. Fundamental operations of modular arithmetic discussed include sign or relative magnitude determination and division, mathematical solution using modular arithmetic, techniques for efficient mechanization of modular arithmetic adders and multipliers, and organization and control of a modular arithmetic computer.

(over)

Numerical analysis studies yielded novel results including the introduction of signed residues, overflow detection techniques, and a division algorithm 3 times as fast as any previously disclosed. A square-root algorithm which is considerably faster than the Newton-Raphson algorithm is discussed.

Implementation techniques are described for trading speed for complexity, programming a computer to extend its range, and design techniques for reducing component counts in adders and multipliers.

A functional simulation of modular arithmetic computation in a conventional computer is described, and statistical data on the operation of the various algorithms is included.

1. Modular arithmetic
2. Residue number system
3. Digital computers
4. Computer logic
5. Mathematics

I. AFSC Project 7062 Task 706205
II. Contract No. AF33(657) 7899
III. Westinghouse Electric Corporation, Air Arm Division, Baltimore, Md.
IV. Contractor's Report No. 1274A
V. Not aval fr OTS
VI. In Astia Collection